

LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-414

A HIGH-LEVEL SIGNAL PROCESSING PROGRAMMING LANGUAGE

James Edward Hicks, Jr.

March 1988

This blank page was inserted to preserve pagination.

A High-Level Signal Processing Programming Language

by

James Edward Hicks, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Computer Science and Engineering

and

Bachelor of Electrical Science and Engineering

and

Master of Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1988

© James Edward Hicks, Jr., 1988

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author
Department of Electrical Engineering and Computer Science
January 15, 1988

Certified by
Arvind
Associate Professor of Electrical Engineering
Thesis Supervisor

Certified by
Dr. Thomas Goblick
Associate Group Leader, Lincoln Laboratory
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A High-Level Signal Processing Programming Language

by

James Edward Hicks, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 1988, in partial fulfillment of the
requirements for the degrees of
Bachelor of Computer Science and Engineering
and
Bachelor of Electrical Science and Engineering
and
Master of Computer Science and Engineering

Abstract

The motivations for an abstract, diagrammatic signal processing language are presented along with a study of the semantics that such a language should have. *D-PICT*, the proposed Digital Signal Processing Pictorial Language, is thoroughly described. *D-PICT* has a diagrammatic representation with a corresponding textual representation. The diagrammatic representation explicitly shows the parallelism available in the procedures defined and is similar to the flow graphs used to depict signal processing algorithms by researchers in that field. *D-PICT* is strongly typed, but type declarations are only required for procedures.

The compiler is able to generate efficient code because the problem domain is limited; the compiler can support signal optimization techniques that a general-purpose compiler could not. The compiler consists of two stages in order to meet the goal of machine-independence. The compiler for *D-PICT* first parses either diagrams or text into dataflow graphs. Transformations are performed on these dataflow graphs to instantiate generic operators and to do machine independent optimizations. Domain specific optimization techniques such as stream optimization are discussed. The second stage of the compiler is machine dependent, and attempts to generate the most efficient code for the target machine.

Thesis Supervisor: Arvind

Title: Associate Professor of Electrical Engineering

Thesis Supervisor: Dr. Thomas Goblick

Title: Associate Group Leader, Lincoln Laboratory

Acknowledgments

I would like to thank my two advisors: Professor Arvind, my thesis advisor; and Dr. Thomas Goblick, my company supervisor; for their confidence in my ability and their guidance and support. I would also like to acknowledge the 6A program and Lincoln Laboratory for providing the opportunity to work with such a high-caliber group of people.

My colleagues made many contributions to this project. At Lincoln Laboratory, Jon Leivent and Bryan Williams helped by critiquing my ideas for the semantics of the language. Ken Traub from the Computation Structures Group at MIT helped in my implementation of the compiler using his Id compiler and Dataflow Compiler Substrate. Jon Leivent, John Delaney, my wife, Sharon, and my parents all helped by slogging through early drafts of this document.

None of this would have been possible without the love and support of my parents throughout my formative years. I would also like to thank my wife, Sharon, for her encouragement and love while I was doing my thesis work.

Contents

1	Introduction	11
1.1	Background Information	11
1.2	Group 21	13
1.3	Research Goal	14
1.4	Organization	15
2	Signal Processing Language Criteria	17
2.1	Deriving Required Abstractions	17
2.1.1	Silhouette Feature Extraction	17
2.2	Modern Language Criteria	19
2.2.1	Procedural Abstraction	19
2.2.2	Data Abstraction	20
2.2.3	Type System	20
2.3	Language Requirements for Signal Processing	21
2.3.1	Diagrammatic Representation	21
2.3.2	Application Specific Optimization	22
2.3.3	Modeling Signals and Images	23
2.3.4	Desired Signal Operations	24
2.4	The Signal Abstraction	25

2.4.1	The Signal Datatype	26
2.4.2	Primitive Signal Operations	30
2.4.3	Composite Signal Operations	31
2.4.4	Reshape Operations	35
2.4.5	Pointwise Operations	36
2.4.6	Reduction Operators	36
3	<i>D-PICT</i>	37
3.1	<i>D-PICT</i> Types	38
3.1.1	Primitive Types	38
3.1.2	Composite Types	39
3.1.3	Vectors and Arrays	41
3.2	Type Declarations	42
3.3	Primitive Operators	44
3.3.1	Scalars	44
3.3.2	Tuples	44
3.4	The Generic Operators	45
3.4.1	The Unary Operators	46
3.4.2	The Binary Operators	47
3.5	Signal Operators	49
3.5.1	Primitive Signal Operators	49
3.5.2	Signal Domain Operators	51
3.6	Signal Combination Operators	52
3.6.1	Union	52
3.6.2	Join	53
3.6.3	Signal Projection Operators	57

3.7	Function Application	58
3.8	Conditional Operators	59
4	The Compiler	61
4.1	Compiler Organization	62
4.2	Program Graph	64
4.2.1	Signal Instructions	65
4.2.2	<i>Join</i> Operators	65
4.2.3	<i>Map</i> Instructions	65
4.2.4	Encapsulating Join Operations	68
4.3	Stream Optimizations	70
4.3.1	Serial Stream Optimization	70
4.3.2	Parallel Stream Optimization	72
4.3.3	Program Graph Optimizations	72
4.4	Conversion to Loop Instructions	74
4.5	Compiler Back Ends	76
4.5.1	TTDA Machine Code	76
4.5.2	Lisp Output	76
4.5.3	Specialized Object Code Generators	78
5	Results	79
5.1	Further Research	79
5.2	Conclusion	79
A	<i>D-PICT</i> Network Examples	81
A.1	First Order Filter	81
A.2	Convolution	81

A.3 Fast Fourier Transform	83
A.4 Long Signal Filtering	84

List of Figures

2-1	A first order IIR filter section.	22
2-2	An Example Signal	28
4-1	The Map Instruction	66
4-2	A join operation.	69
4-3	After conversion to map-instruction.	70
4-4	Serial Stream Optimization	71
4-5	Parallel Stream Optimization	73
A-1	A first order filter section.	82
A-2	Signal Scalor Network	83
A-3	Unit Delay Network	84
A-4	Convolution Network	85
A-5	Fast Fourier Transform	87
A-6	Evens Network	88
A-7	Odds Network	89
A-8	Long signal filter	90
A-9	Segmenter Network	91
A-10	Segment Network	92
A-11	Each segment filterer	93
A-12	Segment adder	94

Chapter 1

Introduction

1.1 Background Information

Signal and image processing applications utilize algorithms that often contain a great deal of inherent parallelism. Unfortunately, there are few methods of implementing such algorithms efficiently without worrying about very low-level implementation details. Many such applications, such as those in the SKETCH [33] system, are implemented on special purpose hardware or written in low-level languages such as C or assembly language. Using these powerful languages the programmer can write very efficient procedures; however, the amount of detail that the programmer must specify often obscures the actual algorithms that are being written. The resulting programs are very difficult to write and debug, and are very machine dependent.

Because most high speed signal processing has been done on special purpose hardware, an abstract signal processing language has never become widely used. Almost every time a machine is designed to process signals efficiently, a new language is designed to program that machine. An abstract signal processing language should be machine independent; one should be able to compile an application written in the language for many different computers without having to make significant changes to the program. Although the language itself should be general purpose, one should be able to compile programs to take advantage of special purpose hardware. The language should have the power to exploit any parallelism present in the algorithms, and it should have the ability to express algorithms while hiding the implementation details.

There have been many attempts at defining a high-level signal processing language, such as SIPROL [9], SRL [15], and others described in “Signal Processing Tools” [10] and “Automatic Generation of Time-efficient Digital Signal Processing Software” [21]. So far, no language of this class has achieved widespread use. Unfortunately, each of the previous high-level signal processing languages has drawbacks that prevent it from becoming widely used.

The SIPROL language, and the KBSP [8], ISP [13] and Sketch [33] environments all support the use of the object-oriented programming paradigm in signal processing. Although each of these systems is being used in the research group in which it was developed, the programs lack portability because there is a very limited number of machines for which these systems are available.

SIPROL is based on Pascal, so it inherits some of Pascal’s limitations: the inability to perform separate compilation, and the over-specification of variable types. A system used for the research or development of algorithms must support this development in a more user-friendly manner. A signal processing language should allow some amount of polymorphism, and its compiler should allow incremental compilation.

In KBSP, ISP and Sketch, where Lisp-based languages are used, type declarations are not required, so the compiler does not have enough information to perform many optimizations. The major problem with these systems is that the compilers are not sophisticated enough to generate procedures that run very efficiently. These systems may have the performance necessary for algorithm development, but one will grow frustrated quickly when trying to run a large test. Signal processing researchers will not move from a low-level language to an abstract one unless the abstract language yields comparable performance. Sketch allows the researcher to achieve this performance by writing parts of the applications in C, but using C offsets some of the transparency and portability gains of a high-level language like Lisp. Even if these languages satisfy the performance criteria of researchers, there is little chance that the programs may be transported without change to an application environment, where much higher performance is required.

“On-line Simulation of Block-diagram Systems” [7] and “High-speed Block-diagram Languages for Microprocessors and Minicomputers in Instrumentation, Control, and Simulation” [16] both describe attempts at a diagrammatic representation for signal processing

algorithms. There is also some precedent for general diagrammatic languages, as in “Dependence Graphs and Compiler Optimization” [17]. None of these diagrammatic languages has appealed to a large programming community — these languages have only been used in the small groups in which they were developed. A diagrammatic representation has greater difficulty denoting constructs such as conditional and loop expressions than a textual language. For signal processing applications diagrammatic languages are natural because the algorithms themselves are often described by diagrams; these diagrams are often easier to understand than the corresponding mathematical expressions. Consequently, signal processing has a head start as far as defining diagrammatic languages, since the diagrammatic representation has been evolving for many years.

BDC [35] lacks a type system, so many optimizations cannot be performed, although current research will probably alleviate this problem. In a computation intensive application such as signal processing, this inability to optimize is devastating. Terepin describes a graphical notation for data flow in [26], but the notation essentially describes register transfers in a special purpose processor, so the notation achieves neither abstractness nor machine-independence.

1.2 Group 21

This thesis is a result of work done in Group 21, Machine Intelligence Technology, at Lincoln Laboratory. One project in Group 21 characterizes acoustic signals of aircraft using signal processing algorithms. Other projects in Group 21 extract features from images acquired by video, laser-radar and doppler-laser sensors in order to identify the objects in the scene.

Currently, Group 21’s applications are implemented using Common-Lisp or C. In the SKETCH [33] system, a signal processing environment developed by Group 21, applications are programmed in a combination of LISP and C. LISP is used to provide a high-level programming interface, while C is used to implement procedures that need high efficiency. Thus C is used to write most of the actual image processing procedures.

Group 21 is developing a parallel processor, the MX-1 [31], to aid the research in image processing and understanding. One of the goals of this project is to enable the researchers to write the image processing algorithms in a high-level language, Dotlisp [18]. By running

a variant of Lisp on a parallel processor, Group 21 hopes to reduce the run time of signal processing experiments enough that complete applications may be written in the high-level language.

1.3 Research Goal

The goal of this thesis is to develop a high-level graphical and textual language for signal processing applications. This language should be powerful enough to express any signal processing algorithm efficiently on a general purpose parallel processor. In the future, this language could be generalized to handle other applications that have synchronous dataflow characteristics. A language such as this which takes a diagrammatic input would be a natural extension to a system like SKETCH which is already visually oriented.

Although this language is very abstract, it is possible to generate efficient machine code because the compiler will be operating in a limited application domain. Signal processing algorithms have many common characteristics that can be encoded into the abstractions of this language. One of these characteristics is the inherent parallelism in signal processing applications; often the same computation is repeated independently on each point of an image or a discrete-time signal. Another characteristic is that different algorithms often need to window or reshape data. The high-level language should allow the user to reshape the data as needed, without introducing any great inefficiency into the final program. Finally, the language needs abstractions to support the different ways of combining an arbitrary set of inputs into a set of outputs.

The language proposed, the Digital Signal Processing Pictorial Language (*D-PICT*), will be incorporated into an interactive algorithm development environment. At the most abstract level, the user will enter algorithms as flow graphs or block diagrams. The workstation will take the explicit data and control flow information and determine implementation information, and then generate object code to implement the algorithm. Each graphical input will correspond to a regular programming language construct that will specify precisely how the data will be shaped and how the program will operate on the data. The compiler will use this information to generate code for the target machine, thus being able to optimize enough to produce code comparable to that written by a person. Most of these

optimizations will be common to all the target machines, so that a majority of the compiler can be common to all machines; only the very back-end of the compiler will have to be specifically designed for each target machine. Once the algorithm has been implemented, one should be able to compile it for many different machines, including special purpose DSP processors. If desired, one should be able to generate specifications for integrated circuits from the *D-PICT* definitions. *D-PICT* will be abstract enough to be used at the algorithm development stage, while being able to generate efficient code for machines with less friendly user interfaces, such as special purpose DSP processors.

1.4 Organization

This chapter provided a brief history of the work that has been done on special purpose high-level languages for signal processing. The next chapter will describe the requirements that a signal processing high-level language (HLL) must satisfy, and the third chapter will define the DSP language, *D-PICT*, which satisfies these requirements. Chapter Four describes a compiler for *D-PICT*, to show the feasibility of implementing an HLL for signal processing. Concluding remarks are made in Chapter Five. Some representative *D-PICT* programming examples are shown in Appendix A.

Chapter 2

Signal Processing Language Criteria

2.1 Deriving Required Abstractions

A useful signal processing language must clearly and concisely support the constructs and algorithms used in signal processing applications. In order to determine the necessary abstractions, a study was made of the applications described in “Silhouette Feature Extraction in Laser-Radar Range Imagery” [29,34], parts one and two, written by Group 21. Other researchers have also studied signal processing applications to determine the basis needed in a high-level language for signal processing; two of these studies are presented in “Signal Processing Primitives for LISP” [32] and “Image Algebra Techniques for Parallel Image Processing” [24].

2.1.1 Silhouette Feature Extraction

One of the applications studied to gather the required abstractions for signal processing is described in “Silhouette Feature Extraction in Laser-Radar Range Imagery: II. Intensity-Cued Region-Based Approach” [34]. This application is a system for the analysis of object silhouettes extracted from laser-radar range and intensity images. The interesting portion of this application is the actual extraction of regions from range and intensity images. This application runs on the SKETCH [33] system, and was written in a mixture of C and Lisp.

The images used in this application are obtained by laser radar: a CO₂ infrared laser that creates pixel-registered intensity and range images. The range images have six meter accuracy; both types of images are 128×60 pixels, but are scaled before processing to 128×128 pixels, because the pixel height of the laser radar images is twice that of the pixel width. The goal in this application is to determine regions in the image that represent the silhouette of the object to be identified. The final result is a set of polygon descriptions that approximate the boundaries of these regions.

The first step is to clean the image to remove errors: to detect and replace *missing values* in the range image. The range image is cleaned using the *mathematical morphology* [25] algorithm described in the paper “Silhouette Understanding System” [30]. Mathematical morphology is a non-linear method for operating on signals that preserves local features in the signals. One may remove high-frequency noise from a signal by low-pass filtering the signal using the convolution operation, but low-pass filtering blurs the edges of sharp features in the signal — mathematical morphology provides a methodology for filling holes while minimally altering outlines. Missing values (MV’s) are either *dropouts* or *outliers*. A dropout is a pixel that has a value between 0 and 4 (where pixels are 8-bit unsigned integers), corresponding to points within 4 range units of the sensor. An outlier is a range pixel p of value r that has no neighboring pixel of value r' in the 5×5 region (ignoring dropouts) surrounding p such that $|r - r'| < 10$. The detection and replacement of MV’s must preserve the continuity of the image. These MV’s are removed by performing a 2-step *shrink*, where any MV which has at least L non-missing 3×3 neighbors is replaced by the average of a pair of points in that neighborhood. Then to counteract the extra replacement of MV’s from the image background, a 3-step *expand* is performed, where any non-missing pixel that has at least L missing $M \times M$ neighbors is converted back to an MV. The shrink-expand process is repeated until the image stabilizes (or until a maximum number of iterations has been performed).

Next, range segments, corresponding to the range values of different objects in the image, are determined from the intensity image and the cleaned range image. Intensity-cued range segmentation is performed as follows. A histogram of the intensity-range image pair is computed by forming an array B such that each point $B[i, r]$ is 1 if and only if there is a pixel in the range image with value r whose corresponding intensity image pixel has value i . N range segments are defined by taking the top n range values that have a value of 1 in

the intensity-range histogram B , starting with the highest intensity value. These n range values denote n range *segments*. Note that these segments of the image are not necessarily contiguous; there may be holes in the segment, or the pixels may be sparsely distributed through the segment.

One of the range segments, or regions, is chosen to be the target region. The least-squares line approximating the major axis of the segment is computed for each range segment. The landscape in an image usually consists of narrow horizontal bands; any region that has large vertical spread is probably an object, not background, so the segment with the greatest standard deviation of the range points from the segment's major axis is considered to be the target range. The target region is defined to be the set of all range pixels that are within some error value of the target range.

The binary target image is formed from the range image by setting all pixels in the target interval to `true` and setting all other pixels to `false`. The target image is cleaned using binary mathematical morphology. The effects of this cleaning are to fill the holes in the target region, to reconnect appendages to the body of the target region and to remove ground clutter from the target region.

Finally, this cleaned binary image is passed on to the rest of the system for polygonal approximation of the region boundaries, feature extraction from the polygons, and then object identification.

2.2 Modern Language Criteria

The signal processing language must satisfy the same criteria as any other modern high-level languages. One of the few studies which explicitly mentions this is *The Representation of Discrete-Time Signals and Systems in Programs* [14].

The details of the program should not obscure the algorithm being implemented. This is the principle of *perspecuity*, which is realized through procedural and data abstraction.

2.2.1 Procedural Abstraction

The language must support procedural abstraction so that programs may be developed hierarchically. Whether the programmer prefers top-down or bottom-up development is

unimportant, the language should support either methodology. The programmer should be able to compile each procedure individually. When a procedure is defined to be substitutable, the compiler should be able to recompile all procedures which call this procedure whenever it is recompiled.

2.2.2 Data Abstraction

The language must support data abstraction also. Data representation information should be hidden by the language, while methods are provided for the creation and manipulation of the data structures.

In the case of signals, there should be a uniform reference mechanism for access to signal elements. This mechanism should gracefully extend to multidimensional signals, so that signal processing algorithms that do not depend on the dimensionality of the signal may be implemented to operate on signals of any rank.

Signal data structures should be immutable, because the abstractions they model are immutable. There should be no history-dependent method for testing the equality of two signals. Signals are equal if and only if they have the same domains and the elements of the signals are equal.

The object-oriented paradigm would allow user-definition of data types that satisfy these criteria. For example the sine curves could be a class of signal, described by three parameters: length, frequency, and phase. One then instantiates a signal by specifying the signal class and the appropriate parameters. Other parameters that could be specified are whether to compute the whole signal, to delay evaluation of individual points, or to delay evaluation of the whole signal.

D-PICT supports the signal abstraction, but there are currently no facilities for user definition of new object types. A more robust type system is needed for that approach, so that efficient code may be generated.

2.2.3 Type System

The signal processing language should be strongly typed, so that the compiler can optimize the object code generated from the diagrammatic algorithm description. Strong typing

also prevents the programmer from getting type errors at run time. With strong typing these errors can be rectified at compile time. The compiler can mark these errors at the point where they occur, rather than having error messages pop up at run time far from the location of the actual mistake. This also prevents the use of programs that have type errors in untested execution paths through the program.

However, putting type declarations throughout the diagram is not only unpleasant but unnecessary. By including a type-inference system [20] one can allow the compiler to *infer* the types of intermediate results from the types of the inputs and outputs of a procedure. The algorithm developer only has to declare the types of the procedure's inputs and outputs when defining the procedure. Moreover, the language should support polymorphism as described in [4,5], so that the programmer can develop a single procedure that will work on objects of many types.

2.3 Language Requirements for Signal Processing

2.3.1 Diagrammatic Representation

A graphical language is well suited for signal processing because one thinks of each function operating on a stream of data and returning another stream. This flow of streams, or signals, can be denoted by arcs in the graph, while nodes denote the functions being applied to the streams. Note that there is no implication of precedence in signals; the stream of data representing a signal must be thought of as pairs of indices and data values that may be generated in an arbitrary order. One should be able to draw diagrammatic programs that closely mimic the flow graphs, such as those in *Digital Signal Processing* [23], used to describe signal processing algorithms.

The semantics of the language can be *dataflow*-like, as in Id-Nouveau [22]. Dataflow is a functional language paradigm that permits fine-grained parallelism. The smallest unit of computation is a dataflow instruction. Each procedure is composed of a *graph* of dataflow instructions wired together. Computation proceeds as data values flow through the graph on *tokens*. Dataflow instructions are *strict*, no instruction may produce an output without having values for all of its inputs (see also Section 2.4.1). An instruction is said to be *triggered* when all of its inputs have tokens. At any particular time, all triggered instructions

may *fire*, or perform their computation, and generate new data tokens. Thus a procedure's available parallelism at any point in the computation is equal to the number of instructions that are triggered at that time.

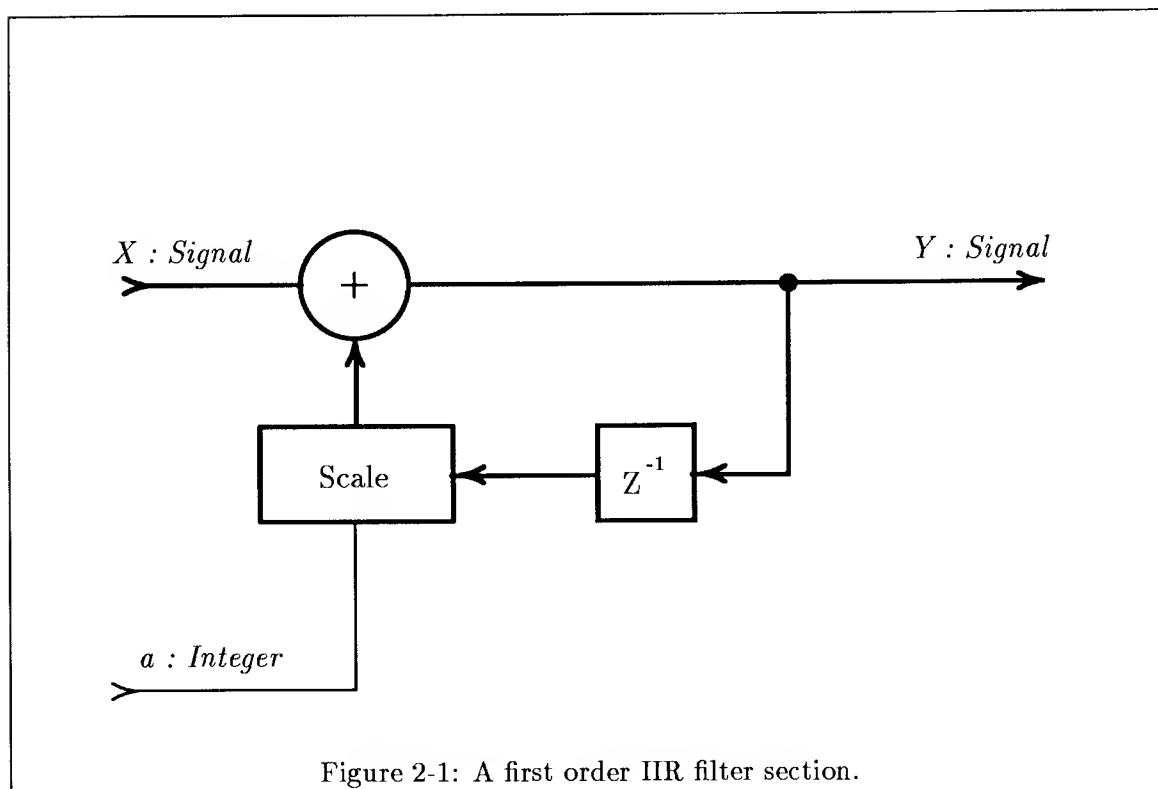


Figure 2-1 shows the flow graph for an infinite impulse response (IIR) filter; each arc denotes the path of a signal or data value. If all arcs of a graph were drawn the same way, then it would be somewhat ambiguous which arcs represent signals and which arcs represent scalars. If the arc labeled **a**, representing the coefficient for the delayed feedback, were drawn the same as the arc for the output signal, **Y**, then there would be confusion as to the type of data traveling along each arc. To avoid confusion, narrow arcs should be drawn to represent the path of scalar values, and wide arcs should be drawn to represent the path of signal values.

2.3.2 Application Specific Optimization

In order to generate efficient code, the compiler must be able to optimize the output code. The compiler should incorporate optimizations specific to signal processing applications. One such optimization is stream optimization, where the optimizer attempts to reduce the

amount of temporary storage allocation.

2.3.3 Modeling Signals and Images

A *signal* is a function that maps from a domain to a range, $\mathcal{R}^n \rightarrow \mathcal{R}$, where the domain usually represents time or distance, and the range represents some physical quantity that is being measured by imperfect and often imprecise instruments. For example, an LP record is the recording of an acoustic signal. It maps continuous time values to continuous amplitude values; when the record is played back one hears the sound that was recorded. A compact disc (CD) is a digital recording of a discrete-time signal formed by sampling a continuous-time acoustical signal. One may reconstruct a continuous-time signal by low pass filtering the discrete-time signal. If CD's were perfect: the samples had infinite resolution and the recording was infinitely long, then the acoustic signal could be reproduced exactly. Sadly, one must take reality into account — the amplitude samples are quantized: only a finite number of bits of resolution are available, and the length of the recording is finite — so the digital recording yields only an approximation of the original sound. Fortunately, this approximation is better than most analog (or continuous-time, continuous-range) recordings, and so is still of great value. These discrete-time, or more generally discrete-domain, digital signals are what *D-PICT* was developed to model and manipulate. Both research and applications use these digital signals to represent sound and images.

In this thesis, *signal* denotes a datatype of *D-PICT* that represents digital signals. A signal can be one-dimensional, as in a discrete-time acoustic signal, or two-dimensional, *e.g.* an image, or more generally n -dimensional. One would like to model signals as potentially infinite streams of data points: an arbitrary function from a domain to a range. The word *stream* in computer science evokes notions of sequentiality, delayed evaluation, and other forms of procrastination, so the term will not be used here, although in a sense a *D-PICT* signal is isomorphic to the concept of a stream that may be infinite or self-referential. Thus a signal conceptualizes a function: $\mathcal{Z}^n \rightarrow T$, where T denotes any type. One can also think of a signal as an array, where the dimensions of the array represent the domain, \mathcal{Z}^n , and the element-type of the array represents the signal's range. Although in reality the dimensions of an array are finite, one should not think about the actual dimensions of the signal being processed unless they are important to the algorithm. The Fast Fourier Transform, for

example, depends on the number of points in the signal being a power of two. One should think of the signal as infinite in extent unless the signal's boundaries are of interest.

The fact that signals may be actually be finite in length and that one may access the signal elements in an arbitrary order prevents the use of a functional stream processing language. In a stream processing language there is a notion of precedence which implies an ordering of the signal element accesses and definitions. This implicit ordering is fine for many applications, but for algorithms which access the signal elements in an arbitrary order this is unacceptable.

Mathematically, signals are immutable; one can create new signals by operating on old ones, but one cannot change the values of a signal once it is defined. The immutability of signals allows signal processing programs to be modeled in a purely *functional* manner, that is, without side-effects. The use of a functional language allows the compiler to generate code for parallel machines with little interaction from the user.

2.3.4 Desired Signal Operations

Primitive Signal Operations These operations perform signal allocation, and the definition and reference of signal elements. It is understood that most users of *D-PICT* will use higher-order operators to create and manipulate signals.

Pointwise Operations The pointwise operations include the standard unary and binary arithmetic operators such as addition and subtraction. Addition of two signals, A and B, means the creation of a new signal, whose domain is the intersection of the domains of the signals A and B, and whose elements are the sums of the corresponding elements of A and B. Other pointwise operations allow the programmer to create a new signal by mapping a function over the elements of a signal.

Signal Combination Operations The union operator takes two signals and produces a new signal by taking the union of the sets of index-value pairs of each signal. The intersection of the domains of the input signals should be the empty set in order for this operation to be well-defined.

One will have to be able to create new signals whose domains are the intersection or cartesian product of the input signals' domains. The new signal elements will

be formed by applying some function to the data value and index of each point in the domain of the newly defined domain. One can actually implement the pointwise operations mentioned above by applying the appropriate arithmetic operation to the data values of each input signal over the intersection of the signals' domains.

Reshape Operations Often signal processing algorithms require the signal to be *reshaped* before being operated on by a procedure. For example when computing the Fast-Fourier Transform (FFT) [23] of a signal in a straightforward recursive implementation of the FFT, one must perform the FFT on the even-numbered data values and also on the odd-numbered data-values, and then combine these two smaller FFT's into the final FFT. Taking only the even-numbered points of a signal is an example of reshaping the signal. Other reshaping operators take the rows or columns of a two-dimensional signal, shift the origin of a signal or up-sample or down-sample a signal by some amount.

Reduction Operations Often one wants to generate a single value from a signal, such as the sum, product, maximum or minimum of the values of a signal. Analogous operations for binary signals are **Some**, where true is returned if any of the values of the signal is true, and **Every**, where true is returned only if every value of the signal is true.

Another type of reduction operator is an accumulation. In the case of the intensity-range histogram defined in the application above, the value returned from the operation is actually an array of values. In *D-PICT* a value of this sort is an **accumulator**. An **accumulator** has dimensions, **d**, an initial value, **iv**, and an accumulator function, **f**. When an **accumulator** is instantiated, each of its points has the value **iv**. Any subsequent write of a value **w** to location **j** of an accumulator **A** causes the location **j** to take the value **f(A[j], w)**.

2.4 The Signal Abstraction

The high-level signal processing language must support operations on **signals** efficiently. The language must have a datatype that embodies the signal abstraction and on which efficient operations may be performed.

The language must allow support for more complex issues that deal with the finiteness of real computers — what to do with missing values, arithmetic overflows (truncate them, make missing-value, or signal an error), and how to handle the boundaries during operations on arrays. Also, the language needs to be able to handle extremely large signals — ones that will not fit in memory all at once — caching parts of the signals will be necessary.

2.4.1 The Signal Datatype

A signal is a non-strict, immutable datatype consisting of at least a domain and a set of pairs of element values and their indices. Signal descriptors should contain the following information about signals:

- **Element Type** The type of the signal’s elements.
- **Dimensions** The actual boundaries of the domain of the signal.
- **Rank** The number of dimensions of the domain of the signal. A signal’s rank may be deduced from its dimensions.
- **Default-Value** If a signal has a default value, then it has an infinite domain.
- **Sampling Information** This would contain the sampling rate of the signal.

Strictness A function f is *strict* in its i th argument if f always diverges, or yields bottom (\perp), when its i th argument diverges. For example, f is strict in its first argument if and only if

$$f(\perp, v) \Rightarrow \perp$$

for all v . A function is said to be strict when it is strict in all of its arguments. An intuitive notion of strictness is that a function is strict if and only if it always requires the value of each its arguments (see also “Strictness Analysis — A Practical Approach” [6]).

Using this intuitive notion of *strictness*, signals are non-strict because one does not always need every signal element in order to instantiate the signal. Let the function $1D_signal$ be

the constructor of one-dimensional signals. $1D_signal$ is of type $((pair(Z \times Z) \times T \times \dots \times T) \rightarrow sig(T,1))$; it takes a domain (an upper and lower bound) and n signal elements, and returns a one-dimensional signal with element-type T . If the domain diverges, then $signal$ diverges:

$$signal(\perp, v_1, \dots, v_n) \Rightarrow \perp.$$

Therefore, $signal$ is strict in its first argument. However, $signal$ will return a valid signal if the domain is defined even if some or all of the elements are undefined:

$$signal((1, n), v_1, \dots, \perp, \dots, v_n) = \text{a-signal}.$$

Therefore, $signal$ is non-strict in the rest of its arguments. The signal data structure is strict in its domain but non-strict in its elements, so signals are non-strict data structures. One can operate on a partially defined signal:

$$fetch(signal((1, n), v_1, \dots, v_n), i) = \begin{cases} \perp & \text{if } v_i \Rightarrow \perp \\ v_i & \text{otherwise} \end{cases}$$

Although $fetch$ is strict in both its arguments, s and i , nearly all of the elements of s could be undefined and $fetch$ would still return a value; the only requirement is that s , i and the i th element of s are all defined. The non-strict signal data structure is similar to the non-strict `cons` described in *Dataflow Architectures* [2].

Although signals are non-strict, the implementation does not have to be *lazy*. An *eager* evaluator computes all expressions as soon as the expression's subexpressions have been evaluated; a lazy evaluator does not perform a computation unless its result is needed by another expression. Lazy evaluation is also described as *demand-driven*, as opposed to *data-driven*, or eager evaluation. Even an efficient lazy implementation [11] is less efficient than an eager one because a lazy implementation requires a closure to be stored in each array slot. When the value of a particular slot of a lazy array is required, then that slot's closure is evaluated, and the result of the evaluation is returned as the slot's value. If *memoization* [1] is being performed, then the closure in the slot is replaced by the result; consequently, the evaluation of a slot's value need only be performed once. In signal processing, one usually operates on the complete signal; there is no unnecessary computation to be avoided in computing a signal, so there is no motivation for a lazy implementation. One should use

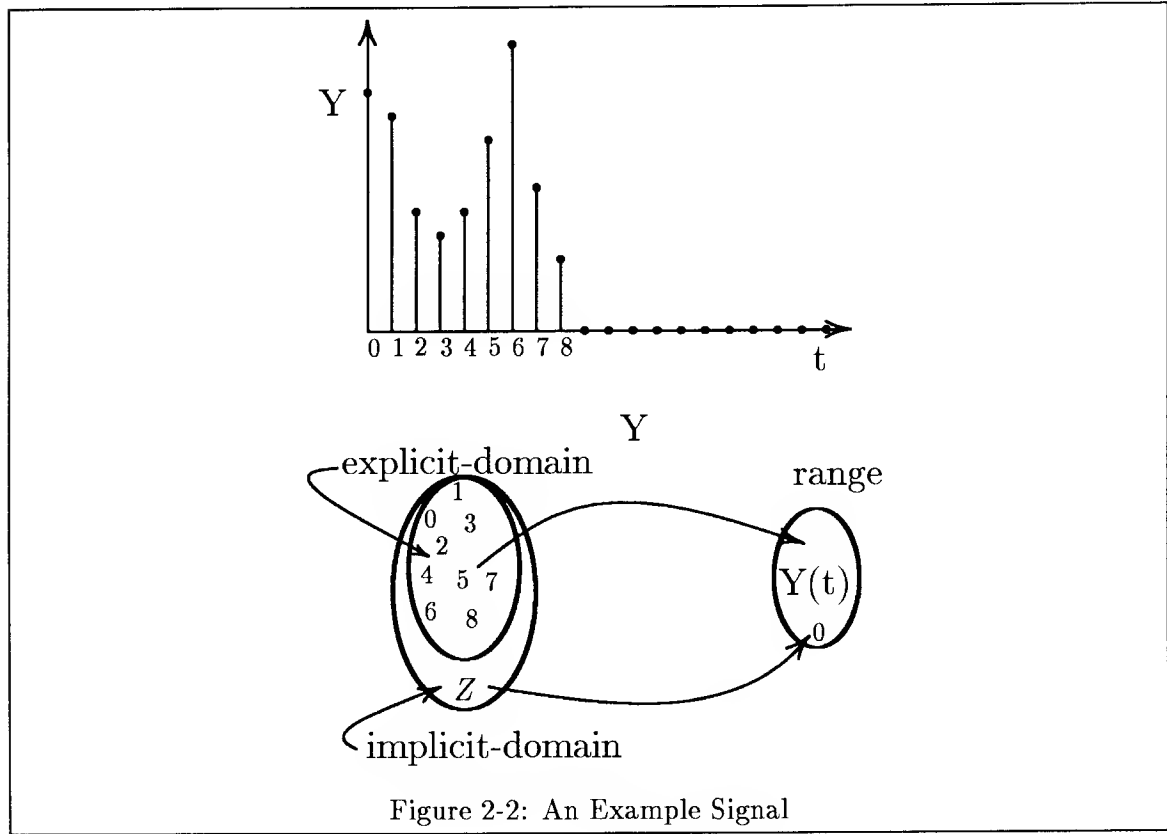
iterative expressions to compute the signal contents and let resource management techniques such as *Loop Bounding* [3] control the amount of available parallelism. If memory cannot hold the complete signal or if there is a high probability that a significant portion of the signal will never be referenced, then a partially lazy implementation would be valuable.

Signal Domains

The domain of a signal is the index space over which the elements of the signal are defined. A signal is essentially a function from the domain to the value space. A signal $Y(t)$ can be represented by the set of pairs:

$$\{\langle t, Y(t) \rangle \mid \forall t \in Domain\}.$$

For example, in Figure 2-2, the signal $Y(t)$ is composed of the pairs:



<i>index</i>	<i>Y</i>
0	10
1	9
2	5
3	4
4	5
5	8
6	12
7	6
8	3

The signal's domain can be made infinite by specifying what value to return when the specified index is outside of the signals *explicit* domain. The signal *Y*'s *explicit* domain is

$$\{i \mid i \in \mathcal{Z} \wedge 0 \leq i \leq 8\}$$

and its *implicit* domain is all other integers, because it has a *default value* of 0.

A continuous-domain signal can have vectors of real numbers (elements of \mathcal{R}^N) as its indices; however, *D-PICT* constrains the indices of signals to be vectors of integers (elements of \mathcal{Z}^N). A one-dimensional signal will have integers as its indices because vectors of length one, the degenerate case for vectors, are just scalar objects of the element type of the vector.

Signal Implementation

Signals are conceptually equivalent to a set of pairs, where each pair consists of an element value (from the signal range) and an index (from the signal domain). However, to implement the algorithms as efficiently as possible, the information will be stored in a different form. A useful representation for a signal is:

```
deftype signal (rank, element-type) =
  struct{
    bounds :    tuple( vector(integer), vector(integer)),
    origin :    vector(integer),
    step :     vector(integer),
    data :     array( rank, element-type ),
```

```

    default-value: element-type
}

```

where **bounds** are the virtual boundaries of the signal, **step** specifies that only every **step**th element of the array is considered, and **origin** is where point $\vec{0}$ of the image maps onto the array. A **signal** is an example of the use of dependent types (for a discussion of types see “On Understanding Types, Data Abstraction and Polymorphism” [5]). The number of elements in the **vector(integer)** depends on the **rank** of the **signal**, the type of **default-value** depends on the **element-type** of the signal, and the type of the **data** array depends on both the signal **rank** and **element-type**. Using this representation, a two-dimensional signal:

```

signal[x,y]

```

translates to:

```

If ((signal.lower-x <= x) and (x <= signal.upper-x)
    and
    (signal.lower-y <= y) and (y <= signal.upper-y))
Then (signal.data)[ (x * signal.step-size + signal.origin-x),
                    (y * signal.step-size + signal.origin-y) ]
Else signal.default-value;

```

This representation allow efficient reshaping of signals. One creates a new signal descriptor in which the origin is shifted in order to perform a translation; in which the step size is increased, to perform down-sampling; or in which the bounds are changed, to window the signal. No copying of the data array needs to be done for these operations.

2.4.2 Primitive Signal Operations

General signal operations must be built up from primitive signal operations. The primitive operations, **Empty-Signal**, **Signal-Ref**, **Signal-Set**, **Signal-Bounds** and **Signal-Default** allow the creation and manipulation of signal descriptors.

The operation **Empty-Signal**((l,h) , d) creates a signal with bounds (l,h) , default-value d , and rank $r = \text{length}(l)$. The signal’s domain is the set of all vectors of length r where each element, e_i , of the vectors in the domain are in the range $l_i \leq e_i \leq h_i$.

The default value of the signal is the value that the signal takes outside of the explicitly specified domain. If d is specified, then the signal has an infinite domain. The values of the signal inside the domain must be defined before they can be referenced by the program. Any references to unspecified values of the signal will be *deferred* until those values are specified. The operation **Signal-Bounds** returns the domain of the signal, and the operation **Signal-Default** returns the default value of the signal.

The operation **Signal-Ref**(s, i) returns the value of signal s at index i . If the value has not yet been written, the **Signal-Ref** instruction is deferred until the value is written. If the index i is outside the explicit domain of the signal, and the signal has a default value, then that value is returned, otherwise an error is signalled. Of course, if the value is never written, then the **Signal-Ref** is deferred forever (and the expression yields \perp).

The operation **Signal-Set**(s, i, v) sets the value of signal s at index i to the value v . If the index is outside the explicit domain of the signal, or if the value of the signal at index i has already been defined, then an error is signalled.

2.4.3 Composite Signal Operations

In the signal processing language, one must be able to specify operations on signals. Procedures that operate on signals as a whole resemble database operations. These operations, as described in *Algebra and Databases* [19], are

- Selection
- Union
- Join
- Projection

One may think of complex signal operations as applying a database transformation to the signal domains and specifying a function to combine the signal elements at each point in the resulting domain in order to create a new signal or scalar value.

Union

The union operation takes two signals and creates a new signal by taking the union of the sets of pairs of each signal.

$$\text{union}(s_1, s_2) \Rightarrow s_3$$

where $s_1 = \{\langle i, d(i) \rangle \mid i \in I_1\}$ and $s_2 = \{\langle i, d(i) \rangle \mid i \in I_2\}$, defines the new signal s_3 , such that $s_3 = \{\langle i, d(i) \rangle \mid i \in (I_1 \cup I_2)\}$. The input signals must have disjoint explicit domains in order for the union to be well-defined. Also, if both signals have default values, then they must have the same default value or an error is signalled.

Selection

The selection operation takes a set of indices and returns the index-value pairs from the signal such that each pair returned has one of the specified indices. If the set of indices has cardinality one, then selection is the **signal-ref** operation.

$$\text{select}(s_1, p) \Rightarrow s_2$$

where $s_1 = \{\langle i, d(i) \rangle\}$ and $p = \lambda \text{index}. \text{boolean}$, produces the signal s_2 such that $s_2 = \{\langle i, d(i) \rangle \mid p(i)\}$.

Join

The *join* database operation (inner-join) provides an extremely flexible method for signal combination. The join operation produces a new signal whose domain is the specified inner-join of the input signals' domains and whose element values are computed by applying a function to the n-tuples of the input signals' element values at each point in the new domain. The general join operation takes a combining function and n signals and *domain specifications*. Each domain specification is a list of d integers, where d is the rank of the associated signal's domain. The integer j_i in the i th position of the domain specification indicates to map the i th dimension of the input signal's domain to the j th dimension of the join operation's domain. If the domains of more than one signal are mapped to the same dimension of the join operation's domain, then that dimension of the join's domain is defined to be the intersection of the specified dimensions of the input signals' domains. The

result of the join is a signal whose domain is equal to the domain of the inner-join operation performed, and whose elements are computed by applying the combining function to the appropriate elements of the input signals. Consider the signals X and Y shown in the following tables:

<i>index</i>	X
(1,1)	a
(1,2)	b
(2,1)	c
(2,2)	d

<i>index</i>	Y
(2,1)	e
(2,2)	f

The domain of signal X is the set $\{(1,1),(1,2),(2,1),(2,2)\}$, and the domain of Y is the set $\{(2,1),(2,2)\}$. If all of the input signals have infinite implicit domains, then the join domain should be the smallest n-cube that encloses the explicit domain of all of the input signals. If any signal's implicit domain is finite, then the join operation's domain is bounded by that signal's explicit domain.

If the domain specification for each signal is the same, then `join` computes the intersection of the domains of the input signals. The expression `Join(foo, X, (0,1), Y, (0,1))` represents the *intersection* of X and Y using the function *foo* to combine the elements of the two signals. The resulting signal has the same rank as the input signals. The following table contains the signal resulting from this intersection.

<i>index</i>	$X \cup Y$
(2,1)	foo(c,e)
(2,2)	foo(d,f)

If none of the integers in the input signals' domain specifications are the same, then the `join` operation computes the cartesian product of the input signals' domains. The expression `Join(foo, X, (0,1), Y, (2,3))` denotes a *cartesian product* of the signals X and Y using the function *foo* to combine the elements of the signals. The resulting signal is of rank four and is shown in the following table.

<i>index</i>	$X \times Y$
(1,1,2,1)	foo(a,e)
(1,1,2,2)	foo(a,f)
(1,2,2,1)	foo(b,e)
(1,2,2,2)	foo(b,f)
(2,1,2,1)	foo(c,e)
(2,1,2,2)	foo(c,f)
(2,2,2,1)	foo(d,e)
(2,2,2,2)	foo(d,f)

Finally, the join operator allows more complicated combination of signals that are neither the intersection nor the cartesian product of the input signals' domains, *e.g.* $Join(foo, X, (1, 2), Y, (0, 1))$:

<i>index</i>	$Join(f, X, (1, 2), Y, (0, 1))$
(2,1,1)	foo(a,e)
(2,1,2)	foo(b,e)
(2,2,1)	foo(c,f)
(2,2,2)	foo(d,f)

Of course, finding an application for this feature may be difficult, but it is comforting to know that the more useful operations, intersection and cartesian-product, are based on semantically sound primitives.

Projection

The projection operation maps one space onto another, reducing the rank of the input space. A useful variation of the database version of this operation allows one to generate a single value or array of values by folding together or reducing the elements of a signal. Thus, a project operation can compute a summation or histogram of the signal elements. *D-PICT* does not allow the general version of projection that can reduce the domain of a signal arbitrarily; a projection of a signal can only produce a single value: a sum or histogram. A histogram is a strict data structure; one cannot access any elements of a histogram until all writes to any element of the histogram have been completed.

Database operators do not allow transitive closure operations, but these operations are useful for higher-level signal operations such as region-labeling and polygonal-approximation. The altered project operation, allowing histogram computation, can be used to generate transitive-closures as well. The accumulator structure created would be a digraph instead of an array.

2.4.4 Reshape Operations

Although one may reshape a signal by operating on its data array, the cost in terms of new signal allocation is prohibitive in any real system. These operations can be performed by operating on the signal descriptors; in most cases this precludes the necessity to copy the data array of a signal's representation.

Windowing may be performed by creating a new signal descriptor which has the same data array but has a new domain. The descriptor contains the new bounds, and has its origin defined to map from the virtual origin of the window to the actual origin of the signal to be windowed.

A signal is shifted by creating a new signal descriptor whose origin has the amount of the shift in each dimension added to that dimension of the origin.

Operations such as convolution map a window across a signal; creating a new signal structure for each window would consume tremendous resources; however, common subexpression elimination and fetch elimination optimizations should cause the bounds and origins of the windows to be present only in loop variables, not in heap storage.

The signal data structure contains a *step* descriptor which is useful for down-sampling a signal. By specifying a step of n , where $n \neq 1$, one can down-sample the signal by n . This means that only every n th element of the signal will be present in the down-sampled signal. The bounds of the signal must also be adjusted accordingly. Up-sampling requires generating a new signal with $m - 1$ zeros between each element of the up-sampled signal, for an up-sampling ratio of m . If the signal descriptor contains sampling-rate information, then this information must also be adjusted when one performs up or down-sampling.

2.4.5 Pointwise Operations

The standard binary operators are built on top of the join operations. A simple intersection of the input signal domains is performed, and the elements from each signal are combined with the appropriate binary operator to form the new signal. For example, signal addition is performed by taking the intersection of the input signals and adding the pairs of signal elements to compute the elements of the output signal.

2.4.6 Reduction Operators

The reduction operators are implemented by performing a projection on the signal, using the specified procedure to compute the scalar or histogram result. In this way, one may find the maximum value of a signal by projecting the signal, using the `max` function to combine signal elements.

Chapter 3

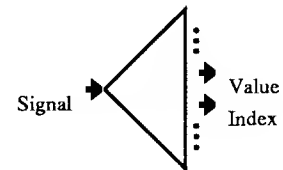
D-PICT

The Digital Signal Processing Pictorial Language, *D-PICT*, is a diagrammatic language for implementing signal processing algorithms. Diagrams, or networks, describe functions that implement the desired algorithms. A network is composed of operators and arcs. *D-PICT* operators behave functionally; the output of an operator depends solely on the inputs to the operator. The arcs in a network replace variables; each arc represents a data path. Signals are immutable objects represented by arrays.

This chapter describes the operators of *D-PICT*. Each operator has an entry with the following form:

Operator-name *args*

Operator-Signature



Each entry of this form describes a *D-PICT* operator whose name is **Operator-name**. In the textual representation, **Operator-name** would be written in the function position, and the **args** would be written in the args position. In the diagrammatic representation, one would instantiate the operator named **Operator-name**, and would wire the desired arcs to the input ports of the operator. The results of the operations are available at the output ports. **Operator-Signature** describes the type of the operator. Normally this is a procedural type expression, $(t_1 \times \dots \times t_m) \rightarrow (t'_1 \times \dots \times t'_n)$, denoting a function. *D-PICT* procedures and operators may operate on cartesian

products of values, denoted by $(t_1 \times \cdots t_m)$, and may also return a cartesian product. The operator shown in this entry is `Elts`, which is described in Section 3.6.

3.1 *D-PICT* Types

Although *D-PICT* is a strongly typed language, one would like to avoid interspersing type declarations throughout the definition of an application. One only has to declare the type of a procedure once, the first time it is instantiated or edited. One never has to declare the types of temporary variables (of course this only applies to the written form of *D-PICT*, because there are no intermediate variables as separate entities in the diagrammatic form). In order to meet this goal, *D-PICT* has a type inference system that requires only the input and output ports of a network to have type declarations. *D-PICT* uses a unifying type inference system *à la* Milner[20] to determine the type of each node in a network. *D-PICT* also allows polymorphic procedures; types that are unknown at compile time are denoted by type variables of the form $*$ or $*n$.

3.1.1 Primitive Types

Void

The `void` type represents an undefined value, and is denoted by \perp . `Void` is not a value that one can use in a program; it represents the value of a signal element before a legal value has been stored. `Void` also represents the type of an expression that returns no value.

Numbers

D-PICT has integers, floats, and numbers, which may be either integers or floating point numbers.

```
6847 : Z      % An Integer
6.341 : F      % A Float
Amplitude : N  % Amplitude may be either an Integer or a Float.
```


Booleans

There are two boolean values:

```
true : B
false : B      % Boolean literals.
Condition : B  % Condition is a Boolean variable.
```

Characters

```
\1 : Char      % A Character
Letter : Char  % Letter is a Character variable.
```

Strings

```
"Signal Processing" : S  % A String
Name : S              % Name is a String.
```

Symbols

All symbols have type “**symbol**” and are written as quoted identifiers.

3.1.2 Composite Types

D-PICT’s composite types are parameterized over all the *D-PICT* types. These are the tuple, procedure, signal, vector and array types.

Tuples

Tuples are heterogeneous conglomerations of objects. The tuple type is represented by the expression `Tuple(*1 × ... × *n)`. Tuples are strict immutable objects. A tuple literal is written as `(a,b,c)`. Tuples are distinct from the cartesian products that *D-PICT* procedures take as inputs and return as outputs.

Procedures

The types of procedures are represented by $t_0 \rightarrow t_1$ for some argument type t_0 and result type t_1 . Note that both t_0 and t_1 may be cartesian products, so that the procedure takes more than one argument or returns more than one value.

```
not : B → B    % Function from booleans to booleans.  
+   : N × N → N % Binary arithmetic operator.
```

Index Type

An index, `index(rank)`, denotes either an integer, for a one-dimensional signal, or a vector of integers, for a multiple-dimensional signal. This type allows *dimensional extensibility*: a procedure that does not depend on the rank of the signals on which it operates is not constrained by the type of the indices.

```
index = Z | tuple(Z × Z) | tuple(Z × Z × Z) | ...  
  
1 : index(1)      % A one-dimensional index.  
(2,4) : index(2)  % A two-dimensional index.
```

The type `index(1)` is equivalent to the integer type, and the type `index(2)` is equivalent to the type of a 2-tuple of integers: `tuple(Z × Z)`, and `index(3)` is equivalent to `tuple(Z × Z × Z)`, and so on.

Signals

The signal datatype was motivated by the goals of *D-PICT*. The language was designed to manipulate signals, and so this type must be very adaptable. Signal types have two parameters: `element-type` and `rank`, and are written `sig(element-type, rank)`. The `element-type` is the type of the data elements of the signal. The `rank` denotes the number of dimensions of the signal; an acoustic signal has rank one, and an image has rank two. The indices of a signal are of type `index(rank)` (either integers or tuples of integers). The bounds of a signal consists of a lower bound and an upper bound. The lower bound is an index defining the lower left corner of the n-dimensional domain of the signal. The upper

bound defines the upper right corner of the signal domain. If the signal is one-dimensional, then the bounds will be a tuple of two integers. The rank and element-type, but not the actual domain, of a signal must be specified at compile time.

```
speech : sig( N, 1)  % A speech signal is one-dimensional.
image  : sig( N, 2)  % An image is a two-dimensional signal.
```

Signals of different rank are distinct types; for example, **speech** and **image** are of different types because **speech** is one-dimensional and **image** is two-dimensional.

***D-PICT* signals** are immutable structures that represent discrete-domain signals. A **signal** has three slots: **data**, **bounds** and **default**. **Data** is an array that holds the element-values of the signal. **Bounds** describes the domain of the array. **Bounds** is a pair of indices representing opposite corners of the domain. **Default** is the signal's default value; if the signal has a default value, then its *implicit* domain is infinite, but its *explicit* domain is still described by **Bounds**.

Signals are non-strict data-structures, in that one can return a signal value that does not have all of its elements determined. In order to maintain the correctness of the computation, one must not be able to access a signal element before it has been defined. Therefore ***D-PICT*** will use an array construct similar to ID I-Structures [22], where the array can be returned as soon as it is allocated, but any read on an array element is *deferred* until a write has been performed on that element. I-Structures are immutable; each location can only be written once. I-Structure semantics guarantee that no matter when a read is performed, the correct value will be returned, and the non-strictness of the data structure allows more space-efficient procedures to be written than with strict, immutable (purely functional) data structures.





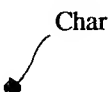
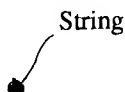

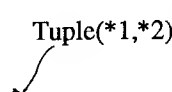

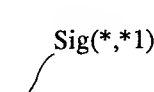
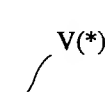
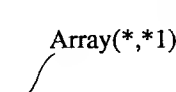
3.1.3 Vectors and Arrays

D-PICT has vectors and arrays that resemble Lisp vectors and arrays. These are declared with **vector(element-type)** or **array(element-type, rank)**. A vector is type-equivalent to a one-dimensional signal, and an array of rank n is type-equivalent to an n -dimensional signal, assuming that the element-types of each are also type-equivalent.

3.2 Type Declarations

These assertions are included in order to give a pictorial representation for the type of an arc in a network. They do not perform any operation except annotation. Only the arcs from a *D-PICT* network's input ports and to the network's output ports need to be annotated with type declarations; the type inference module of the compiler will determine the type of all other arcs.

Each of the type declarations is shown in the following table.

integer 	float 
number 	boolean 
char 	string 
symbol 	istuple 
proc 	signal 
vector 	array 

There is no **void** type declaration because no arc may have type **void**.

The **tuple** declaration takes n parameters $t_1 \dots t_n$, and declares the type of its input and output to be **tuple**(t_1 , \dots t_n). A tuple is an immutable structure with unnamed slots. It is similar to a vector, being implemented as an array, but a tuple does not have to be homogeneous; the type of each slot may be different.

A **proc** declaration takes two parameters, argument type and return-type.

The **signal** declaration takes two parameters, element-type and rank, and declares that

the value it receives is a signal with that element-type and rank.

The **array** declaration takes two parameters, element-type and rank, and declares that the value it receives is an array with that element-type and rank. The compiler currently does not handle operations on arrays.

The **vector** declaration takes one parameter, element-type, and declares that the value it receives is a vector of that element-type. This is just a specialization of the **Array** declaration; a vector's rank is always one.

3.3 Primitive Operators

3.3.1 Scalars

Scalars may be numbers, symbols, booleans, characters, or strings. They are treated as immutable objects by *D-PICT*.

Literal l

$typeof(l)$

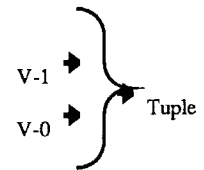


Literal is really a schema for introducing literal values into a network. The particular **literal** shown is for the integer 0; the value of any literal will be displayed above the body of the operator. The value produced by **literal** is the value 1.

3.3.2 Tuples

Tuple $v_0 \cdots v_{(n-1)}$

$*_0 \times \cdots \times *_{(n-1)} \rightarrow \text{tuple}(*_0 \times \cdots \times *_{(n-1)})$

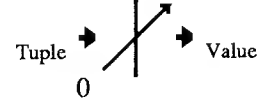


Tuple is a data structure constructor: given n elements of type $*_0, \dots, *_{(n-1)}$; it

produces a tuple whose components are of type $*_0, \dots, *_{(n-1)}$. **tuple** creates a tuple of length n containing values $v_0, \dots, v_{(n-1)}$.

Tuple-Fetch $t\ i$

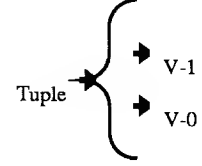
$\text{tuple}(*_1 \times \dots \times *_n) \times Z \rightarrow *_i$



Tuple-Fetch returns the i th element of the tuple \mathbf{t} . The number at the lower-left corner of the operator denotes which element of the tuple is being fetched.

Untuple t

$\text{tuple}(*_0 \times \dots \times *_{(n-1)}) \rightarrow *_0 \times \dots \times *_{(n-1)}$



Untuple takes a tuple \mathbf{t} and returns each element of \mathbf{t} on a separate port.

3.4 The Generic Operators

The basic arithmetic operators are *generic*; they can be applied to numbers, signals of numbers or tuples of numbers. They are actually schema that expand into the code to perform the correct operations. Each operator is defined by a function $\mathbf{f}:(T_1 \times T_2 \rightarrow T_3)$. If the operation is applied to arguments of type T_1 and T_2 , then the code to call the function \mathbf{f} is generated.

One may map a function $\mathbf{f}:(T_1 \times T_2 \rightarrow T_3)$ across two signals using **map_signal** which is of type $((T_1 \times T_2 \rightarrow T_3) \times \text{sig}(T_1, n) \times \text{sig}(T_2, n)) \rightarrow \text{sig}(T_3, n)$ and which maps \mathbf{f} across the two input signals to produce an output signal.

One may also map the function \mathbf{f} across two homogeneous tuples to generate a homogeneous tuple. Let $n\text{-tuple}(T)$ denote a tuple of length n whose elements are all of

Code generated:	Argument types:
$(f\ a\ b) : T_3$	$a : T_1, b : T_2$
$(\text{map_signal}\ f\ a\ b) : \text{sig}(T_3, n)$	$a : \text{sig}(T_1, n), b : \text{sig}(T_2, n)$
$(\text{map_n_tuple}\ f\ a\ b) : n\text{-tuple}(T_3)$	$a : n\text{-tuple}(T_1), b : n\text{-tuple}(T_2)$

Table 3.1: Instantiation of Generic Operators

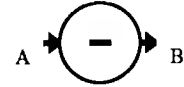
type T . Then the function $\text{map_n_tuple} : ((T_1 \times T_2 \rightarrow T_3) \times n\text{-Tuple}(T_1) \times n\text{-Tuple}(T_2)) \rightarrow n\text{-Tuple}(T_3)$, where n is the length of the tuples, generates the new homogeneous tuple by mapping f across the input tuples.

A *generic* use of function $f : (T_1 \times T_2 \rightarrow T_3)$ will be instantiated by the compiler as follows: The generic operators in this section are instantiated as described in Table 3.1. The generic operator descriptions will therefore talk only about the function f which describes the operator's operation; the signature, or type, of these operators is the signature of the function f . The `unary_map_signal` and `unary_map_n_tuple` are used to implement the unary generic operators in a fashion similar to that of the binary generic operators.

3.4.1 The Unary Operators

– a

$N \rightarrow N$



Negate performs arithmetic negation.

$\neg a$

$B \rightarrow B$

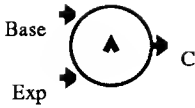


Not performs boolean negation.

3.4.2 The Binary Operators

Arithmetic Operators $a\ b$

$$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

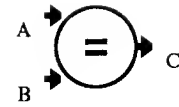


Each of the binary arithmetic operators has the same signature. The operator shown is **exp**, which raises **a** to the **b**th power. For each of these operators, **a** is the first operand, and **b** is the second. The other arithmetic operators: addition, subtraction, multiplication, and division, are shown in the following table.

<div>add</div>	<div>sub</div>
<div>mul</div>	<div>div</div>

Relational Operators $a \ b$

$$N \times N \rightarrow B$$



The relational operators take two numbers as input and yield a boolean. The relational operators are shown in the following table.

eq 	ne
lt 	gt
le 	ge

Boolean Operators $a \ b$

$$B \times B \rightarrow B$$



The boolean operators perform the standard logical operations from booleans to booleans. The boolean operators: and and or, are shown in the following table.

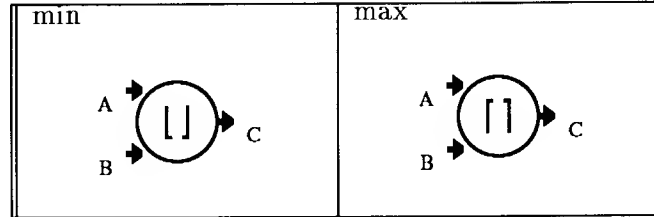
and 	or
---------	--------

Minimum and Maximum $a\ b$

$N \times N \rightarrow N$



Maximum returns the input that is greater in value; **minimum** returns the input that is lesser in value.

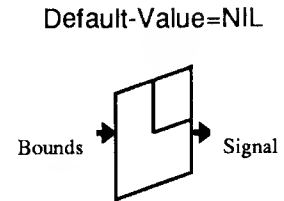


3.5 Signal Operators

3.5.1 Primitive Signal Operators

Empty-Signal b

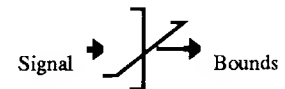
$\text{tuple}(\text{index}(\text{rank}), \text{index}(\text{rank})) \rightarrow \text{sig}(\perp, \text{rank})$



Empty-Signal creates an empty signal with bounds b . The **Empty-Signal** has a single parameter describing the default value of the signal. If the default value is specified, then the element type of the signal is the same as the type of the default value.

Signal-Bounds s

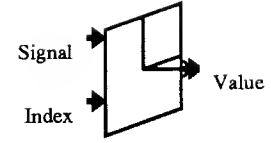
$\text{sig}(*, \text{rank}) \rightarrow \text{tuple}(\text{index}(\text{rank}), \text{index}(\text{rank}))$



Signal-Bounds returns a tuple containing the upper and lower bounds of the *explicit* domain of signal s .

Signal-Ref $s\ j$

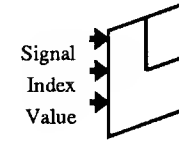
$\text{sig}(*, \text{rank}) \times \text{index}(\text{rank}) \rightarrow *$



Signal-ref returns the value of signal s at point j . If $s[j]$ has not been written yet, then the read is deferred until the write occurs. If the index j is not in the explicit domain of s , then the default value is returned. If the signal has no default value, then an error is signaled.

Signal-Set $s\ j\ v$

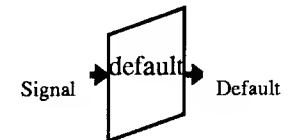
$\text{sig}(*, \text{rank}) \times \text{index}(\text{rank}) \times * \rightarrow \perp$



Signal-set changes the j th value of signal from \perp to v . If the value was not previously \perp then an error results, thus if the j th element of the signal s had already been written, an error would occur.

Signal-Default s

$\text{sig}(*, \text{rank}) \rightarrow *$



Sdefault returns the default value of the signal s , the value of s outside of its explicit domain.

3.5.2 Signal Domain Operators

The following operate on the domains of a signal, without changing the signal's values.

Translate $s \ j$

$$\text{sig}(*, \text{rank}) \times \text{index}(\text{rank}) \rightarrow \text{sig}(*, \text{rank})$$



Translate takes a signal s and creates a new signal that has the same values but whose domain is shifted in each dimension by the corresponding element of index j .

Reflect $s \ a$

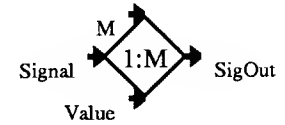
$$\text{sig}(*, \text{rank}) \times \text{index}(\text{rank}) \rightarrow \text{sig}(*, \text{rank})$$



Reflect creates a new signal whose values are the same as the input signal s , but whose domain is the domain of s reflected over each axis whose element of the index a is 1.

Up-Sample $s \ j \ v$

$$\text{sig}(*, \text{rank}) \times \text{index}(\text{rank}) \times * \rightarrow \text{sig}(*, \text{rank})$$



Up-Sample creates a new signal from s by adding points of value v between each element of s . Each element of index j specifies the amount to up-sample each dimension of s .

Down-Sample $s\ j$

$$\text{sig}(*, \text{rank}) \times \text{index} \rightarrow \text{sig}(*, \text{rank})$$



Down-Sample creates a new signal from s by deleting points from signal s . Each element of index j specifies the amount to down-sample each dimension of s .

3.6 Signal Combination Operators

The signal operators allow one to operate on the elements or domain of a signal. For most signal processing algorithms, operations similar to relational database operations are sufficient. These operations treat signals as streams of data points, where a data point is an index paired with a value. In this way one can take the *union* of two signals, one can *join* signals, one can *project* signals, and one can *select* elements from signals. Each of these operations has an analogue in signal processing operations. Unlike database operations, there is no notion of precedence in the processing of signals.

3.6.1 Union

Union $s1\ s2$

$$\text{sig}(*, n) \times \text{sig}(*, n) \rightarrow \text{sig}(*, n)$$



Union creates a new signal that is the union of the two input signals. The input signals must have adjacent but distinct domains. The signals must have disjoint explicit domains; the union of two overlapping signals is not well-defined. If the signals have infinite *implicit* domains, then the two signals must have the same default value. Taking the union of two signals whose domains have different rank or whose domains do not have a common edge is also undefined. Domain rank must be known at compile time, but the actual boundaries do not have to be known until run time. **Union** is analogous to concatenating signals, extended to handle multiple dimensional signals.

3.6.2 Join

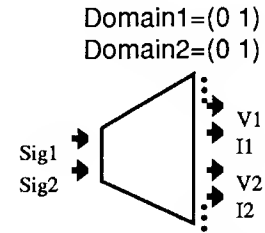
The database join operators: join, intersection and cartesian product, are actually represented by pairs of operators in *D-PICT*. A join operation is formed by a join-driver operator, a join-output operator and the portion of network enclosed by these operators. The driver-collapse pair can be likened to parentheses that separate a subnetwork from the rest of a network. The join-driver specifies the input signals on which to operate and the domain of the join operation to be performed. The join-output operator specifies what sort of object to produce as the result of the join operation: a signal, a scalar (from a projection), or an accumulator. The driver-collapse pair essentially maps a subnetwork over each of the elements of the join of the input signals and collects the values into a result.

Join Drivers

The five join driver operators: **Join**, **Elts**, **Intersect**, **Cart** and **Over-Domain**, specify the domain of the join operation. These operators also specify what values are injected into the body of the join for each iteration.

Join $a_1 \ u_1 \ a_2 \ u_2$

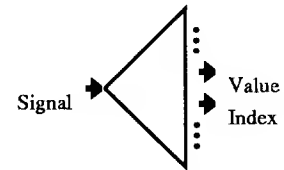
$$\begin{aligned} & \text{sig}(*_0, n_1) \times \text{index}(n_1) \times \text{sig}(*_2, n_2) \times \text{index}(n_2) \\ & \rightarrow *_0 \times \text{index}(n_1) \times *_2 \times \text{index}(n_2) \end{aligned}$$



Join is the general join iteration driver. The vectors u_1 and u_2 specify how to perform the inner-join on the domains of the input signals as described in Section 2.4.3. For each point in the domain of the join, a value from each signal and an index from each signal are produced.

Elts $s \theta$

$$\text{sig}(*_0, n) \rightarrow *_0 \times \text{index}(n)$$

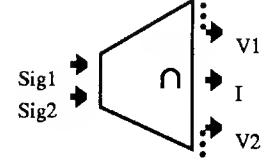


A specialization of join that drives iteration over a single input signal. For each point

in the *explicit* domain of the input signal, **Elts** injects a value and an index into the body.

Intersect *s0 s1*

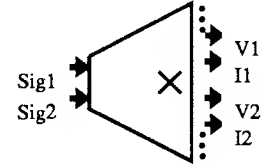
$$\text{sig}(*_0, n) \times \text{sig}(*_2, n) \rightarrow *_0 \times \text{index}(n) \times *_2$$



The **Intersect** operator is a specialization of join. Its domain is the intersection of the domains of the input signals. The input signals must have the same rank. For each point in the intersection of the domains of the input signals, a value from each signal and an index will be produced.

Cart *s0 s1*

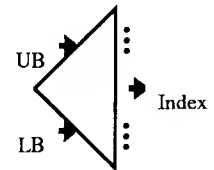
$$\begin{aligned} &\text{sig}(*_0, n_0) \times \text{sig}(*_2, n_2) \\ &\rightarrow *_0 \times \text{index}(n_0) \times *_2 \times \text{index}(n_1) \end{aligned}$$



The **Cart** operator is another specialization of join. Its domain is the cartesian product of the domains of the input signals. For each point in the cartesian product of the domains of the input signals, a value and index from each input signal will be produced.

Over-Domain *lb ub*

$$\text{index}(n) \times \text{index}(n) \rightarrow \text{index}(n)$$



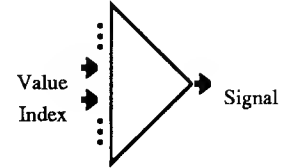
The **Over-Domain** operator specifies the domain bounded by **lb** and **ub**. For each point in this domain, an index will be injected into the join body. This operator is useful for creating signals from scratch.

Join Outputs

These operators: **Collapse**, **Accumulate** and **Project**, specify what type of object to create as the result of the join operation.

Collapse *v j*

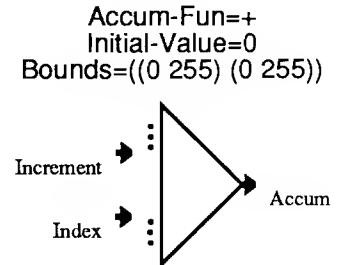
$$* \times \text{index}(\text{rank}) \rightarrow \text{sig}(*, \text{rank})$$



The **Collapse** operator produces a signal with the same domain as that of the join operation. Each point at index *i* in the output signal is defined by the value *v* that was given to the collapse operator. The current compiler only allows the direct connection of the index input from an index output of the loop driver. For a cartesian product the indices must be concatenated. A future version of the compiler should be robust enough to handle operations on the indices presented to the **collapse** operator.

Accumulate *inc index*

$$* \times \text{index}(\text{rank}) \rightarrow \text{sig}(*, \text{rank})$$



The **accumulate** operator specifies that the join operation will create an accumulator (or histogram) as output. This operator has the following parameters:

Accum-Fun This parameter specifies the function with which to accumulate the values. This function must be associative or the accumulated result could vary nondeterministically.

Initial Value This parameter specifies the initial value of each element of the accumulator.

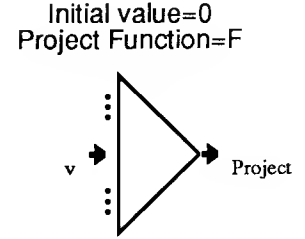
Bounds This parameter specifies the domain of the accumulator created.

The accumulator is a join output operator that creates a signal with dimensions **bds**. Each element of the accumulator starts with the value **Initial Value**. For each point

in the domain of the join operation, the accumulator adds `inc` to the element specified by `Index` using the function `Accum-Fun`.

Project v

$*_1 \rightarrow *_0$



The **Project** operator specifies that the join operation domain will be projected onto a scalar value. This operator has the parameters

Initial Value This is the starting value, of type $*_0$, of the projection output.

Fold Function This function, of type $*_1 \times *_0 \rightarrow *_0$, is used to project the iteration values.

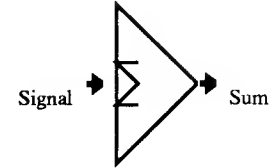
The **Project** operator produces a single value from the values supplied by the join drivers, folding the values together with the function `Fold-Function`, starting with the initial value `Initial-Value`.

3.6.3 Signal Projection Operators

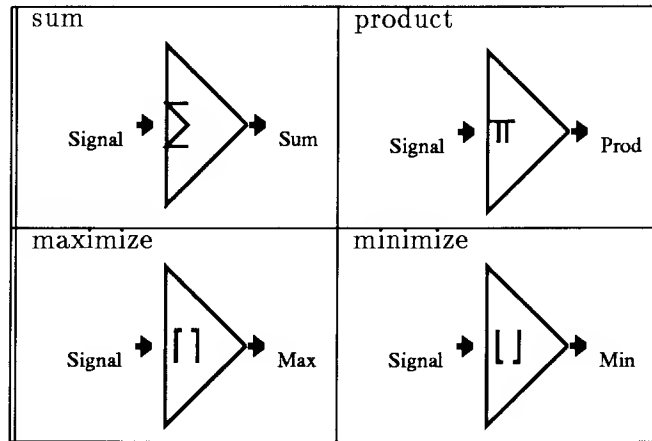
The next six operators are implemented using an `Elts` join driver followed by a `Project` join output operator.

Arithmetic Projections s

$\text{sig}(N, \text{rank}) \rightarrow N$

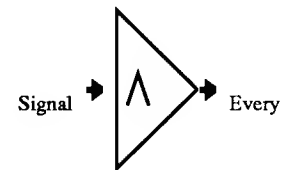


Sum returns the sum of all of the elements of the signal, **Product** returns the product of all of the signal's elements. **Maximize** returns the maximum signal element, and **Minimize** returns the minimum signal element.

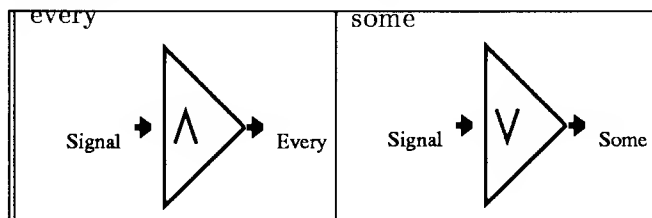


Boolean Projections s

$\text{sig}(B, \text{rank}) \rightarrow B$



Every returns **true** if and only if every element of signal s is **true**, and **Some** returns **true** if any element of s is **true**.

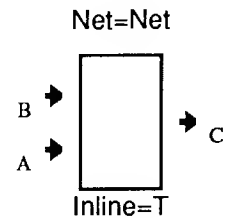


3.7 Function Application

A *D-PICT* function is described by a network. *D-PICT* allows hierarchical development of networks. The net operator defines the invocation of a *D-PICT* network within a network definition. Each net is composed of the network, the net name, and the functional type of the net. Networks are kept in a Net Library once they have been defined. If a net returns multiple values, the procedure it defines returns a tuple of values; however, one may access each of these values on the output ports of a net instance operator in another network's definition.

Net $i_1, \dots i_n$

type specified in net library.



The net operator has the parameters:

Net Name The name of the net to be instantiated.

Inline Specify whether or not to compile the network inline.

The *D-PICT* compiler looks up **net** in the net library, and determines the function type from there. If **Inline** is true, then the compiler finds the net's dataflow graph in the net library entry, and performs in-line substitution. When an instantiation of the **Net** operator is created, the editor instantiates a **Net** operator with the correct number of input and output ports.

Input-Port

*



The **Input-Port** operator represents the value of one of the net's arguments. There will be one of these operators in a network for each argument of the procedure represented by the network.

Output-Port v

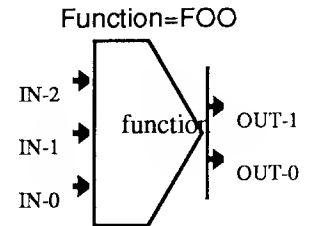
$* \rightarrow \perp$

Output \Rightarrow

The **Output-Port** operator takes as input a value to return as a result of the computation of a net. If a net contains more than one **output-port**, then a tuple of values is returned. However, each of these values may be accessed on the output ports of a **net** schema instantiated in a *D-PICT* network.

Function $i_1 \cdots i_n$

type specified in parameter.



The **Function** schema has two parameters:

Fun The name of the function to apply.

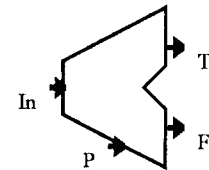
Type The type of the procedure. The network editor determines the number of inputs and outputs from this parameter.

This schema denotes the application of a Lisp function in the network. If the Id compiler back end is being used, then this denotes an Id function application.

3.8 Conditional Operators

If $val\ pred$

$* \times B \rightarrow * \times *$

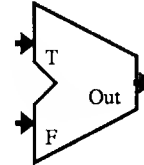


The **If** operator is used to implement conditional expressions. This operator must be paired with the complementary operator, **Fi**. This operator does not actually produce

two values; the type $(* \times B) \rightarrow (* \times *)$, merely denotes that the **If** operator has two outputs; only one of these outputs will produce a value, depending on the value of the **pred** input when the operator is triggered.

Fi *true-val false-val*

$* \times * \rightarrow *$



This operator is used to merge the output of a conditional subnetwork. Only the value corresponding to the triggered arm of the conditional will be output. Similarly to the **If** operator, **Fi** does not take two inputs; it is triggered whenever there is a value on either arm of the input. Because **Fi** must always be paired with **If** its behavior will always be deterministic.

Chapter 4

The Compiler

The *D-PICT* compiler is based on the ID Compiler for the MIT Tagged-Token Dataflow Architecture (version II), as described in [28,27]. The Dataflow Compiler Substrate (DFCS), upon which the ID compiler is based, allows a compiler to be composed of a set of modules, each of which performs some stage of the compilation. DFCS also provides abstractions for creating and manipulating dataflow graphs. Another of its features is support for reading and writing dataflow graphs on input-output streams. DFCS was an invaluable tool for the development of the *D-PICT* compiler.

With DFCS one can define several versions of a compiler, by using different sets of compiler modules. Thus one can define several *D-PICT* compilers: one to generate Lisp output, one to generate Dotlisp output, and one to generate Tagged-Token Dataflow Architecture (TTDA) machine graphs. The front end of each compiler consists of the same modules, but each version of the compiler uses different modules for its back-end, so that the desired object format is generated. When one of the compilers is invoked, DFCS sets up a pipeline of the desired modules, and pushes the source language through the pipeline to create the object code.

Although the ID compiler was indispensable for the development of a compiler for *D-PICT*, it is really a side-light. The ID compiler ties the *D-PICT* system to a Lisp-based computer. In order to make *D-PICT* widely available, there would have to be compilers available for it that are written in a more accessible language such as C. The complete *D-PICT* system should be ported to a wide variety of machines, such as UNIX machines

with X WINDOWS and personal computers with graphics support.

4.1 Compiler Organization

The *D-PICT* compiler is composed of several modules, which will be briefly described here.

Net-Parser The **Net-Parser** module type-checks a *D-PICT* network and then transforms it into a dataflow program graph.

Instantiate-Generic-Operators This module finds all generic dataflow graph instructions that take signals or tuples as inputs and produce signals or tuples, and replaces these instructions with a piece of graph that takes apart the signal or tuple inputs, applies the instruction to each pair of data elements, and creates a new tuple or signal as output.

Collect-Loops The **Collect-Loops** module parses a join construct from a program graph. A join is composed of some *join drivers* that control the domain of iteration and access the signal data elements on which to operate, and some *join output* operators that define how to collect the resulting values. This module encapsulates the join operation in a **Map** instruction. The **Map** instruction has explicit information which allows stream optimization to be performed.

Map-to-Loops The **Map-to-Loops** module transforms a **Map** instruction into a function call to one of the procedures, **Map_n**, where *n* denotes the rank of the iteration space. The **Map_n** procedure takes a signal domain, a map function, and a project function, and applies the map function to each index in the range of bounds. Then **Map_n** folds the results using the project function.

Call-Substitution The **Call-Substitution** module¹ finds all calls of substitutable, or *inlinable*, procedures in the graph. If these applications have all of the required arguments, then these applications are open-coded in the graph.

Fetch-Elimination This module¹ finds all references, or **Tuple-Fetch** instructions, to slots of a structure that are wired directly to the output of the **Make-Tuple** instruction

¹These modules are part of the ID compiler documented in Ken Traub's thesis[27].

that creates the structure. If possible these references are rewired directly to the source of the value that is stored in the appropriate structure slot. If all of the references to a structure allocation are eliminated in this manner, then **dead-code-elimination** can completely remove that structure allocation code.

Cse-and-Hoisting This module¹ performs *common subexpression elimination* and *code-hoisting* optimizations. These two optimizations were included in a single compiler module because they have synergistic effects. Code hoisting attempts to move code out of encapsulators, such as **conditional** and **loop** expressions. Then common subexpression elimination may find more optimizable code, which may trigger more of both kinds of optimization.

Dead-Code-Elimination This module¹ eliminates all instructions whose outputs are not wired to any other instruction and which do not perform side-effects. Function applications may not be eliminated, because they may fill structures as a side-effect.

Inlinable-Definitions **Inlinable-Definitions**¹ stores in the external symbol table the optimized program graphs of procedures that have been declared to be substitutable.

Constant-Propagation **Constant-Propagation**¹ finds all instructions with constant inputs, creates a new constant value by applying the instruction to its inputs, and replaces the instruction by this value.

Transform-Make-Tuples This module transforms **Make-Tuple** instructions into a form for which the **Lisp-Assembler** is able to generate synchronization. Instead of creating an empty tuple and then performing tuple-store instructions, the compiler uses the strict **vector** instruction, which fills a vector with values once all of the instruction's inputs have been computed.

Signals-and-Triggers This module¹ adds *signal* and *trigger* arcs to the program graph for synchronization purposes so that resource managers will know when to deallocate the associated activation tag.

Lisp-Assemble This module transforms dataflow graphs into Lisp code. Dependencies are sorted so that values are produced before any attempt is made to use them. Parallelism is achieved via DotLisp.

Generate-Machine-Graph In this module¹ the machine-independent program graph is transformed into a *machine graph*, with instructions specific to the MIT Tagged-Token Dataflow Architecture.

Peephole This module¹ performs peephole optimizations on the machine graph. One of the most important optimizations is *trigger elimination*, reducing the overhead of synchronization when it is unnecessary.

Fan-out TTDA limits the number of inputs to which any instruction output may be wired to two inputs. The **Fan-out** module¹ adds identity instructions to fan the outputs of overloaded instructions to all of the desired instruction inputs.

Stream-Assembler or File-Assembler **Stream-Assembler**¹ writes the machine graph to an output stream, from which it is loaded into GITA, the Graph Interpreter for Tagged-Token Dataflow Architecture. The **File-Assembler**¹ writes the machine graph to a file in one of the Compiler Input Output Base Language (CIOBL) formats.

4.2 Program Graph

A *dataflow graph* is composed of a collection of *instructions*. Each instruction has an opcode and a given number of inputs and outputs, and can be connected to other instructions in the dataflow graph by directed *arcs*. Arcs *wire* the output of one instruction to the inputs of an arbitrary number of instructions. Arcs can also be *annotated* to give more information about the *tokens* that will flow along them.

A *program graph* is a type of dataflow graph whose instructions are fairly complex (when compared to a machine graph). Instructions are not limited in the number of inputs or outputs, and instructions are not constrained to be strict; firing rules may allow instruction outputs to produce tokens even when some of the inputs have not received any tokens.

The *D-PICT* compiler has several program graph instructions not present in the Id compiler. Naturally when TTDA machine graphs are generated from these program graphs all of the program graph instructions must be replaced by their TTDA equivalents.

The first several modules of the *D-PICT* compiler use the type information gained by the type-inference system. The **Net-Parser** module annotates each arc with a type

declaration. This type information is preserved through all the modules in which it is used. However, in the modules after the **Map-to-Loops** module, the type annotations are gradually lost because the Id compiler does not presently use type information. Future versions of the compiler may use the type information to perform other optimizations.

4.2.1 Signal Instructions

All of the primitive signal operators are represented in the program graph by their own instructions. Most of these operators could have been represented by procedural applications, but having separate instructions for these operators makes graph transformation easier to perform.

There are instructions for **Empty-Signal**, **Signal-Ref**, **Signal-Set**, **Signal-Default**, and **Signal-Bounds**, each of which has the same number of inputs and outputs as the operators of the same name. There are also instructions for the **Reflect**, **Union**, **Up-Sample**, **Down-Sample** and **Translate** operations.

4.2.2 Join Operators

The signal combination operators, denoted by join driver-output operator pairs, are represented in the program graph by similar instructions. The **Elts**, **Intersect**, **Cart**, **Join** and **Over-Domain** instructions correspond to the join driver operations. The join output instructions are composed of **Project**, **Collapse** and **Accumulate**.

4.2.3 Map Instructions

Once the network has been parsed into a dataflow graph, the join driver-output instruction pairs are transformed into a **Map** instruction. The map instruction is an encapsulator instruction similar to the Id *loop* program graph instruction. An encapsulator is an instruction that has inputs and outputs that connect to the rest of the dataflow graph, and also a graph *encapsulated* inside itself whose only inputs and outputs are connected to its internal outputs and inputs.

The map instruction is shown in Figure 4-1. It is very complex compared to most program graph instructions. There will be a **constant-signal** input for each input to a

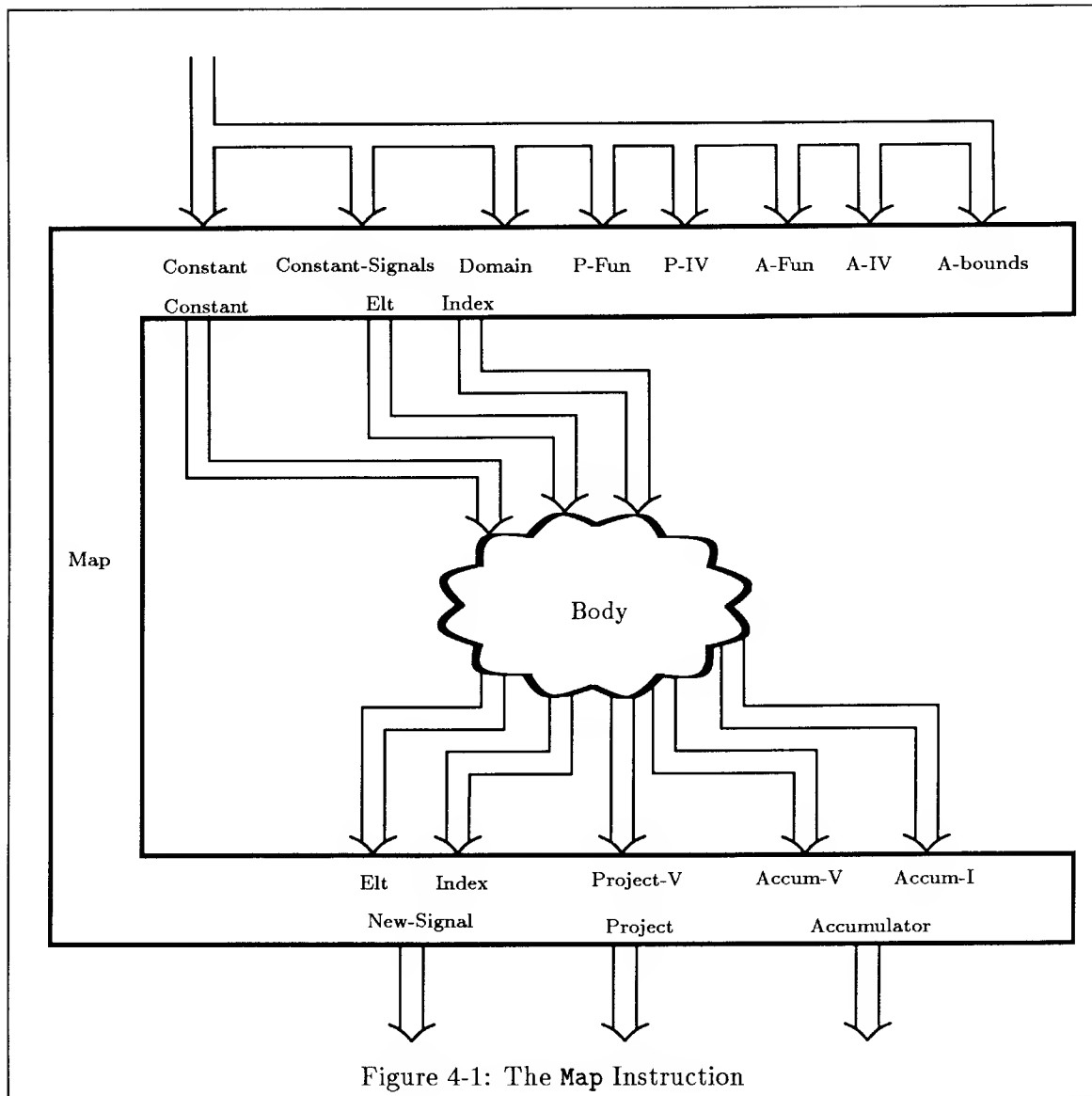


Figure 4-1: The Map Instruction

Elts, **Intersect**, **Cart** or **Join** instruction in the network being compiled. The domain of the map instruction is determined by taking the domain inputs for each constant signal, and the domains of each constant signal, and performing the inner-join of the domains specified by the domain inputs. For each point in the domain of the map instruction, the **Value** and **Index** outputs of the interior of the map instruction will produce tokens containing the value of each of the input signals at that point in the domain, and the index for each of the input signals at that point in the domain.

If any inputs to the *body* of the map construct are constant over the domain of the map operation, then these will flow through the **Constant** inputs of the Map instruction. Each iteration of the Map body will receive a token for each of the constants.

For each **Collapse** instruction in the program graph, there will be a **new-signal** output from the map instruction. All of the new signals produced by a single map instruction will have the same domain. Each new signal produced will have a **value** (**Elt**) and **Index** input in the interior of the map instruction.

For each **Project** instruction in the program graph, there will be a **project-function** (**P-Fun**) and **proj-initial-value** (**p-iv**) input to the Map instruction, and a **proj-value** input in the body of the map instruction. The map instruction initializes each project iteration variable, **pvar**, to the appropriate **proj-initial-value**, and for each point in the domain sets the next value of **pvar** to the value of **P-Fun**(**pvar**, **proj-value**). Finally a value, **Proj** is produced from the map instruction.

The **Accumulator** instructions are treated much the same as the **Project** instructions, except that there is one more input, **Accumulator-Bounds** (**A-Bounds**), for each signal-accumulator instruction. The **Accumulator-Bounds** determines the dimensions of the accumulator signal, **A**, produced by the map instruction. The accumulator is initialized to be an array (signal) of dimensions **Accumulator-Bounds** where each element has the value **a-iv**. For each point in the domain of the map operation, the body produces a value to accumulate, **accum-v**, and the index, **A-Bin**, of the bin in which to accumulate the value; the bin **A[accum-i]** will take the value **A-Fun**(**A[accum-i]**, **accum-v**), where **A-Fun** is the accumulate function.

4.2.4 Encapsulating Join Operations

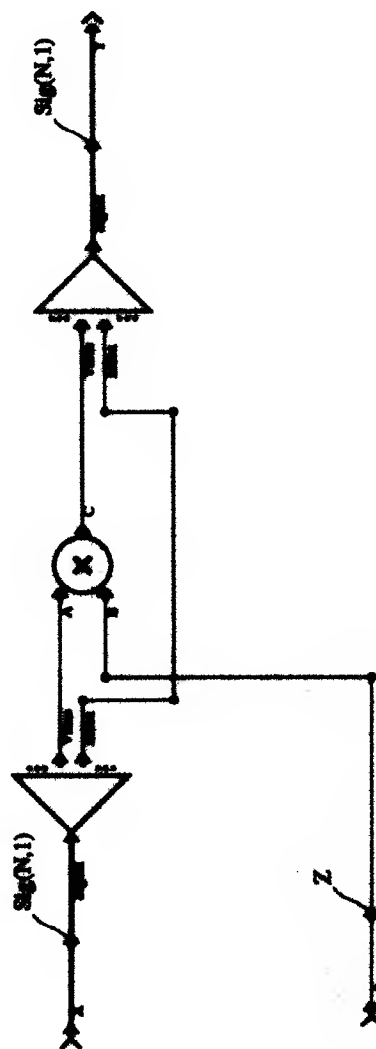
The module `Collect-Loops` determines which instructions in a dataflow graph should be encapsulated in map instructions and performs the encapsulation. It also finds any inputs to these instructions that do not flow through join driver instructions (`Elt`s, `Intersection`, `Join`, or `Cart`), and wires these arcs through the `constant` ports of a map instruction.

In order to find the portion of the graph which is a `map` body, `Collect-Loops` determines the *iteration depth* of every instruction in the graph. An instruction's iteration depth is the maximum depth of any arc wired to the instruction's inputs. The outputs of a join driver instruction are one level deeper than the maximum input depth of that instruction, while the outputs of a join output instruction are one level less deep than the maximum input depth of the output instruction.

All connected instructions whose iteration depths are equal are encapsulated inside a map instruction. This involves finding the join-drivers, join-outputs and loop-constants, and rewiring their inputs and outputs to a map encapsulator. Instructions at deeper levels of iteration are encapsulated in map instructions inside map instructions.

Loop constants are arcs of iteration depth N that are wired to the inputs of instructions of iteration depth $N + 1$. These arcs are constant with respect to the domain of the $N + 1$ depth map instruction. When the body is encapsulated in a map instruction, these arcs will be wired to the constant inputs of the map, and the constant output (from the interior surface of the map) will be wired to the instruction to which the arc was originally wired.

Figure 4-2 shows an example network containing a join operation. Input arcs `X` and `A` and output arc `Y` all have iteration depth of 0. The `Elt`s operator therefore has iteration depth of 1, so the input `A` to the multiplication operator has depth of 1 and finally the multiplication operator and the `collapse` operator have iteration depth of 1. Note that the `B` input to the multiplication operator is of depth zero, so this is a loop constant. Figure 4-3 shows this network after conversion to a map-instruction. The `Def` instruction which wraps the dataflow graph for the procedure is not shown.



SCALE

Figure 4-2: A join operation.

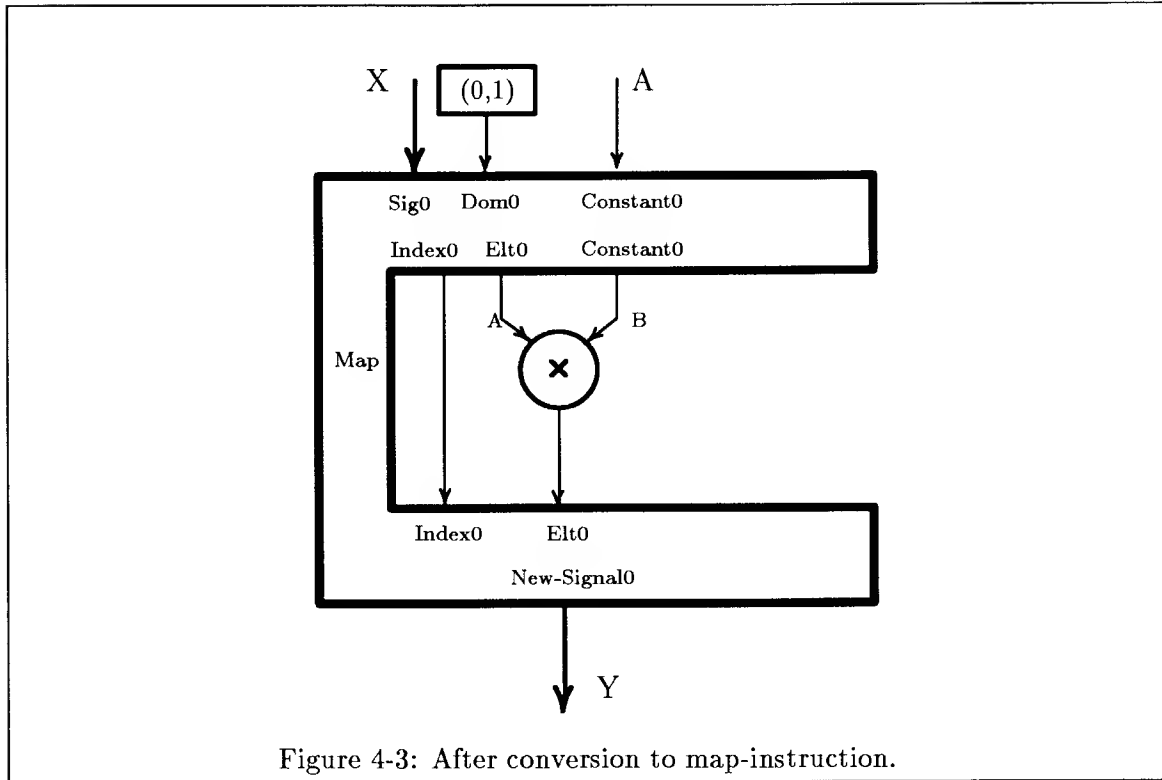


Figure 4-3: After conversion to map-instruction.

4.3 Stream Optimizations

Once the map constructs have been encapsulated in map instructions, one should be able to perform some amount of stream optimization. One would like to reduce the amount of temporary storage allocated. Stream optimization is an \mathcal{NP} -complete problem in general, but the map encapsulator instructions explicitly yield information about usage that allows us to perform some relatively simple optimizations.

4.3.1 Serial Stream Optimization

Signals that are produced and then only used to produce other signals or reduced to produce a scalar value or histogram need never be placed in structure memory. In a case like this there will be a map instruction whose **new-signal** output is wired only to **constant-signal** inputs of other map instructions. A copy of the body of the first map instruction can be wired into the body of the second map instruction, as shown in Figure 4-4. This is an example of serial stream optimization, because the map instructions are wired in series.

If the first map instruction feeds several other map instructions, then putting a copy

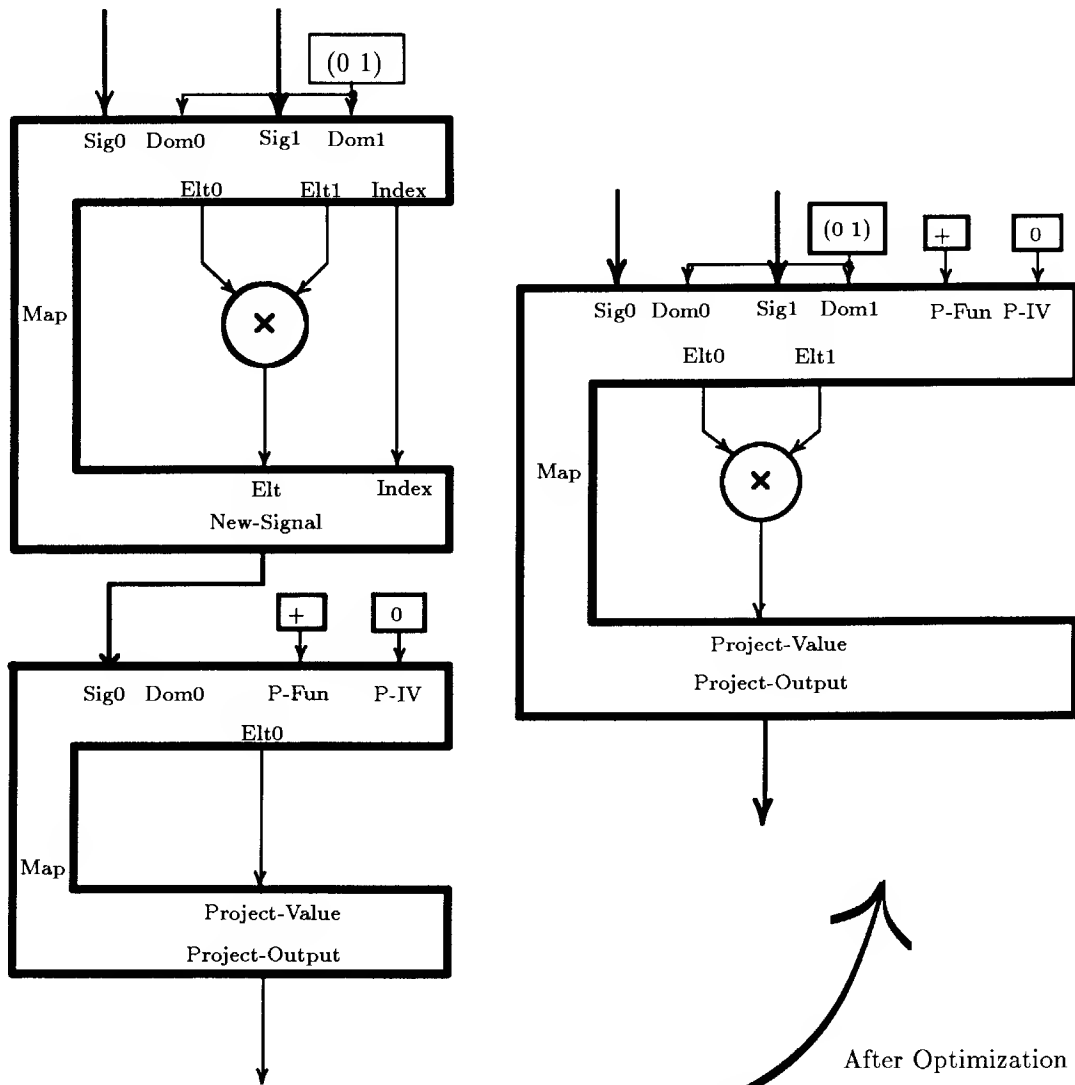


Figure 4-4: Serial Stream Optimization

of the first map instruction's body into each of the receptor maps causes a tradeoff of the time consumed versus the space conserved. The compiler may use a heuristic based on the number of map inputs to which a new-signal output is wired whether or not to perform the serial stream optimization.

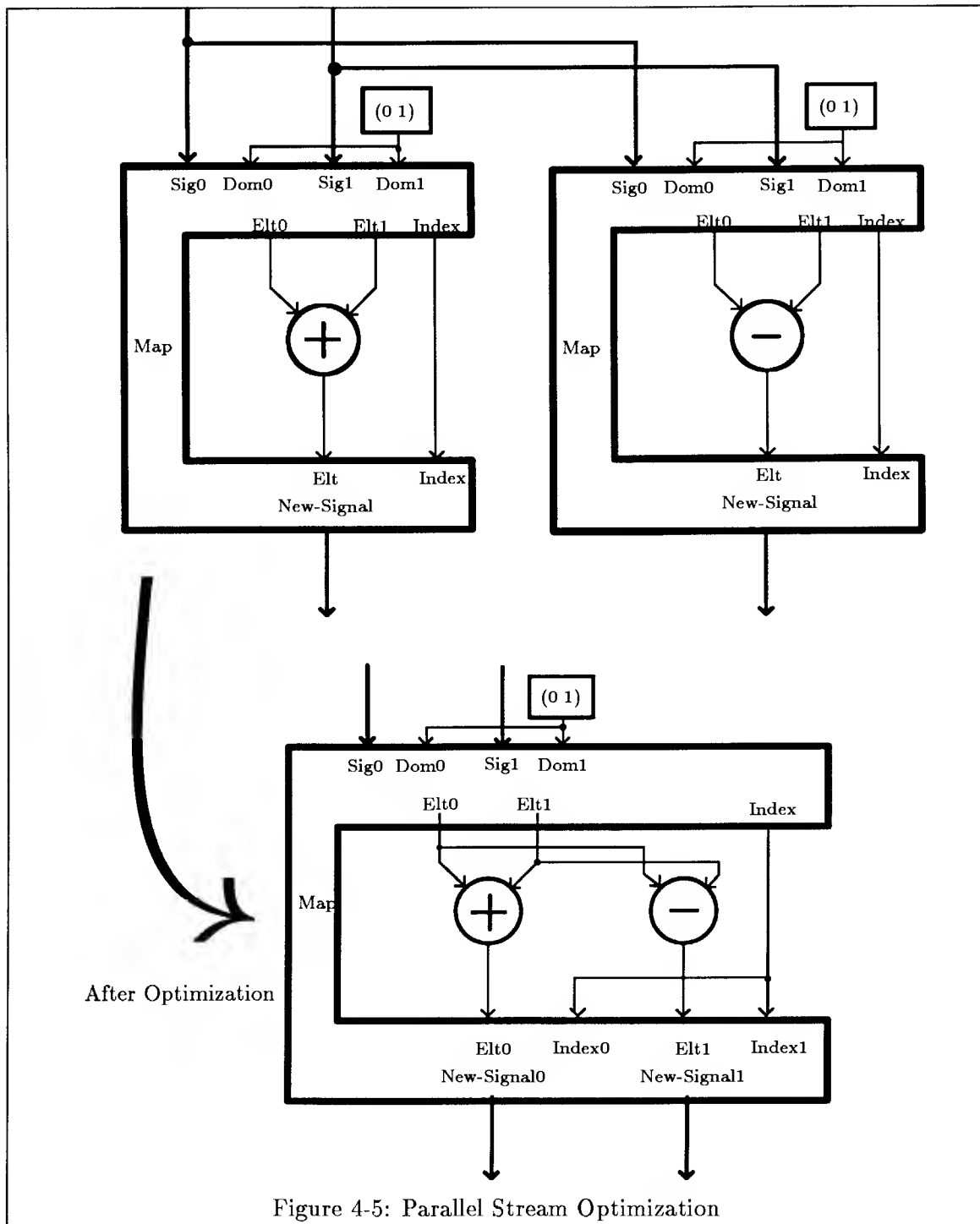
4.3.2 Parallel Stream Optimization

If the program graph contains two map instructions that take the same signals as inputs, and operate over the same domains, then these map instructions may be combined into a single map instruction that operates over the same signals and domain, but contains the bodies of both maps and produces the union of the outputs of these map encapsulators. This optimization is a parallel stream optimization, because the maps would normally operate on the signals simultaneously. By combining the maps, the compiler reduces the overhead used in iteration of the body over the domain. In addition, there may be common subexpressions within the new body that can be eliminated. The parallel stream optimization closely resembles common subexpression elimination, but must be performed by a separate module that has knowledge about signals and domains.

This optimization depends somewhat on the target machine. In a machine where the granularity of parallelism is a loop expression, the loops formed from parallel *map* instructions would be kept separate so that there would be more parallelism available in the object code.

4.3.3 Program Graph Optimizations

The program graph is then pushed through optimization modules that perform the other optimizations: call substitution, fetch elimination, common subexpression elimination, code hoisting, and constant propagation. These optimization modules were all written by the members of the Computation Structures Group at MIT. These modules should come before the *map* instructions are converted to *loop* instructions, so that the program graph saved by the `Inlinable Definitions` module are partly optimized.



4.4 Conversion to Loop Instructions

After any possible (by a conservative estimate) stream optimizations have been performed, the *D-PICT* program graph must be transformed into a graph acceptable to the Id compiler. This is so that optimization modules from the Id compiler can be used unchanged in this compiler. The module Map-to-Loops converts the map instructions to Id program graph constructs.

The body of each map instruction is treated as another procedure. Since the Map body is completely encapsulated by the map instruction, no lambda-lifting [12] needs to be performed, and the body can be defined as a procedure at the top level. The code blocks formed from the map bodies are declared to be inlinable.

After the procedures for the map bodies are compiled, the maps themselves are replaced by two function calls and some graph glue. An application to one of the `Compute_DomainN` procedures is inserted in the graph, where N is the number of constant signals wired to the map. This procedure application computes the domain of the map instruction.

From each map instruction, two new procedures are created. These two procedures are passed as arguments to a function which performs the actual iteration.

The first procedure takes as arguments each of the constants, input signals, empty signals, accumulators, and indices used in the body of the map. Each constant arc in the body will be rewired to an argument output of the procedure's *def* instruction. The index output sources from `signal-intersect`, `signal-cart`, and `signal-join` iteration drivers will be replaced by instructions that make tuples of the correct indices; this index will also be used as the input to a `signal-ref` instruction that will access the value of the input signal at that point. The new-signal index and value inputs of iteration output operators will be wired to `signal-set` instructions that will actually perform the side-effect of filling the new signal. A similar transformation is performed on the accumulate value and index inputs of accumulator operators. The body procedure will return a tuple of the values that are to be folded. The following is the body procedure for the example in Figure 4-2.

```
(defun scale-map-0 (constant-0 constant-signal-0
                  new-signal-0 index-0)
  (i-store (signal_data new-signal-0)
    (+ (signal_origin new-signal-0)
```

```

      (* (signal_step new-signal-0)
         (- index-0 (signal_lb new-signal-0))))
(*
  (i-fetch (signal_data constant-signal-0)
    (+ (signal_origin constant-signal-0)
      (* (signal_step constant-signal-0)
         (- index-0
            (signal_lb constant-signal-0)))))
    constant-0))
(values :void))

```

This body procedure returns `void` because there are no projections. `I-store` takes an array, an index, and a value, and stores the value in the location of the array denoted by the index, and `i-fetch` takes an array and an index, and returns the element of the array denoted by the index. These procedures follow I-structure semantics.

The second procedure, the *project procedure* is much simpler. It takes two tuples as arguments, the first being the tuple of the previous partial sum and the second being a tuple of the values to be summed. There will be one element in the tuples for each `Project` operator in the net. This procedure will apply the appropriate project function to each pair of tuple elements to generate the element in the output tuple. The output of this procedure is a tuple of the new partial sums. Here is the project procedure for the example in Figure 4-2.

```

(defun scale-project-0 (accum value)
  (values value))

```

Since no projections are performed in this example, this project procedure does not do anything.

The actual iteration of the map body over the domain is produced by inserting into the program graph an application of one of the `MapR` procedures, where R is the rank of the map instruction's domain. There is a `MapR` defined for each rank up to $R = 6$, *e.g.* `Map1`, `Map2`, *etc.* . These functions take a domain and two procedures. The first procedure is actually a closure of the body procedure carried with the constants, constant signals, empty signals, and accumulators from the map instruction's inputs and outputs. The second procedure is the `project` procedure. The `MapR` thus fills and accumulates new signals as a side effect of calling the body procedure on each index in the domain, and

performs the projection of values by calling the `project` function on the results of the first application. The only values returned by `Map_R` are the projected values.

After the `map` instructions have been converted to `loop` instructions, call substitution, fetch elimination, common subexpression elimination, code hoisting, and constant propagation should be performed again, to optimize the graph containing loop instructions. The call substitution optimization is vital to the `map` to `loop` conversion, since both the body and the projection portions of the `join` operation are encapsulated in `inlinable` procedure definitions. After the graph has been converted to a standard Id program graph, it is ready to be passed to the back end of the compiler.

4.5 Compiler Back Ends

4.5.1 TTDA Machine Code

When TTDA machine graphs are being generated, the optimized graph is operated on by the rest of the Id compiler back end, which transforms the program graph to a machine graph, adds signals and triggers, limits the fan out of each instruction to two, and assembles the machine graph.

4.5.2 Lisp Output

When Lisp output is desired, the compiler replaces all `make-tuple` and `tuple-store` instructions with `vector` instructions. This is so that the compiler can tell when the corresponding `tuple-fetch` instructions have been triggered (after both the `make-tuple` and `tuple-fetch` instructions have fired). The compiler orders the instructions so that when executed sequentially values will be computed before they are accessed. If an instruction output is used more than once, a local variable is allocated to store that value. Loop instructions are transformed into `prog` statements with loop initialization, test, and loop variable *nextification*. The simple example shown in Figure 4-2 will be compiled to the following Lisp code:

```
(defun scale (x675 a676)
  (let ((v692 :unbound)
        (empty-signal693 :unbound))
```

```

(loop695 :unbound)
(loop696 :unbound)
(loop697 :unbound))
(tagbody
  (setf v692 (signal_bounds x675)
        empty-signal693 (empty-signal v692 ()))
  (multiple-value-setq (loop695 loop696 loop697)
    (let ((loop-constant-0 :unbound)
          (loop-constant-1 :unbound)
          (loop-constant-2 :unbound)
          (loop-constant-3 :unbound)
          (loop-var-0 :unbound)
          (loop-var-1 :unbound)
          (loop-var-2 :unbound)
          (i-store694 :unbound))
      (tagbody
        (setf loop-constant-0 (nth 1 v692)
              loop-constant-1 empty-signal693
              loop-constant-2 x675
              loop-constant-3 a676
              loop-var-0 (nth 0 v692)
              loop-var-1 :void
              loop-var-2 ())
        next
        (unless (<= loop-var-0 loop-constant-0)
          (go finally))
        loop-body
        (i-store (signal_data loop-constant-1)
                  (+ (signal_origin loop-constant-1)
                     (* (signal_step loop-constant-1)
                        (- loop-var-0
                           (signal_lb loop-constant-1))))))
        (*
          (i-fetch (signal_data loop-constant-2)
                    (+ (signal_origin loop-constant-2)
                       (* (signal_step loop-constant-2)
                          (- loop-var-0
                             (signal_lb
                               loop-constant-2))))))
          loop-constant-3))
        nextify
        (setf loop-var-0 (+ loop-var-0 1)
              loop-var-1 :void
              loop-var-2 ())
        (go next)
        finally
        (values loop-var-0 loop-var-1 loop-var-2))))
(values empty-signal693)))

```

The body and project procedures have been open coded into the `scale` procedure. This code is very low-level Lisp to illustrate that a back end for C could easily be created.

The code generated by this module is totally sequential. For the MX-1 parallel processor `DotLisp` code will be generated so that medium-granularity parallelism may be achieved. The loop expressions will be rewritten to allow parts of the loop to operate on different processors.

4.5.3 Specialized Object Code Generators

Although a Lisp machine is a good development environment, one may want to run the *D-PICT* program on a special-purpose computer in order to have the program run very fast. One could have special back ends for the *D-PICT* compiler that produce code for dedicated DSP processors. One could even have a back end that interfaced with silicon compiler software to produce an integrated circuit that directly implemented the algorithm. In this way, *D-PICT* allows performance scalable from that necessary for algorithm and development to all levels of performance needed in the actual application.

Chapter 5

Results

5.1 Further Research

Further research remains to be done on the type system. Strongly-typed languages prevent type errors from happening at run-time. The programmer can debug much more efficiently because these errors are found at compile time when the exact location of the mistake is easier to determine. A language that could catch static typing errors and not burden the programmer with type declarations would be very useful. *D-PICT* takes one step towards that goal, but its type system is not powerful enough yet. Another worthwhile goal would be to add user-defined data types to *D-PICT*.

There needs to be more study of the error handling mechanisms for problems such as arithmetic overflow and underflow, and more work should be done on explicit storage management and stream optimization in the compiler.

5.2 Conclusion

The goal of this thesis was to show that a language such as *D-PICT* is desirable and feasible. Working with the *D-PICT* system has shown this to be true. Because the actual system was not the main thrust of the thesis, the construction of the user interface was not emphasized, and so the system is often difficult to use.

If a novel language like *D-PICT* is ever to become widely used, it must not only be

demonstrated to be desirable and feasible, but there must also be implementations of it on many popular computers with output to many popular programming languages. The system currently runs on Texas Instruments' EXPLORER Lisp machines only, but in the future there will be *D-PICT* implementations on other machines as well.

I believe that there is a need for an abstract signal processing language such as I have described. In order for signal processing work to be feasible on a general parallel processor, one must be able to program applications at the algorithmic level, without having to worry about the details of parallelizing the application. An abstract signal processing language will aid people developing signal processing applications, because they will be able to enter programs at the algorithmic level while still getting efficient code from the compiler. The high-level program should contain enough information so that programs compiled for a parallel processor will have a large amount of parallelism wherever it is available. In addition, the resulting signal processing programs will not be tied to a single machine or architecture, so they can be compiled to run on whatever machine the programmer is using, including machines with special purpose hardware intended for specific applications.

Appendix A

D-PICT Network Examples

A.1 First Order Filter

Figures A-1, A-2 and A-3 show the network definitions for a first order infinite impulse response (IIR) filter section. This program is an example of a simple operation on a potentially infinite input signal. For a measure of the perspecuity of *D-PICT*, compare the *D-PICT* network in Figure A-1 with the block diagram of a first order IIR filter section in 2-1. The two diagrams are almost identical, except that all *D-PICT* operators are constrained to take their inputs on the left and produce their outputs on the right. Most block diagram languages describe this kind of algorithm very well, while array based signal processing languages usually have trouble, because there is inherent sequentiality due to the feedback loop in the algorithm.

A.2 Convolution

The definition of a two-dimensional convolution procedure is shown in Figure A-4. This definition is suitable on relatively short signals, although in principle it would work on any size signal. Note that there is little that needs to be changed to allow this procedure to operate on signals of other ranks. The input to the `Reflect` operator, and the `Convolution-Bounds` procedure are the only things which limit this to operation on two-dimensional signals.

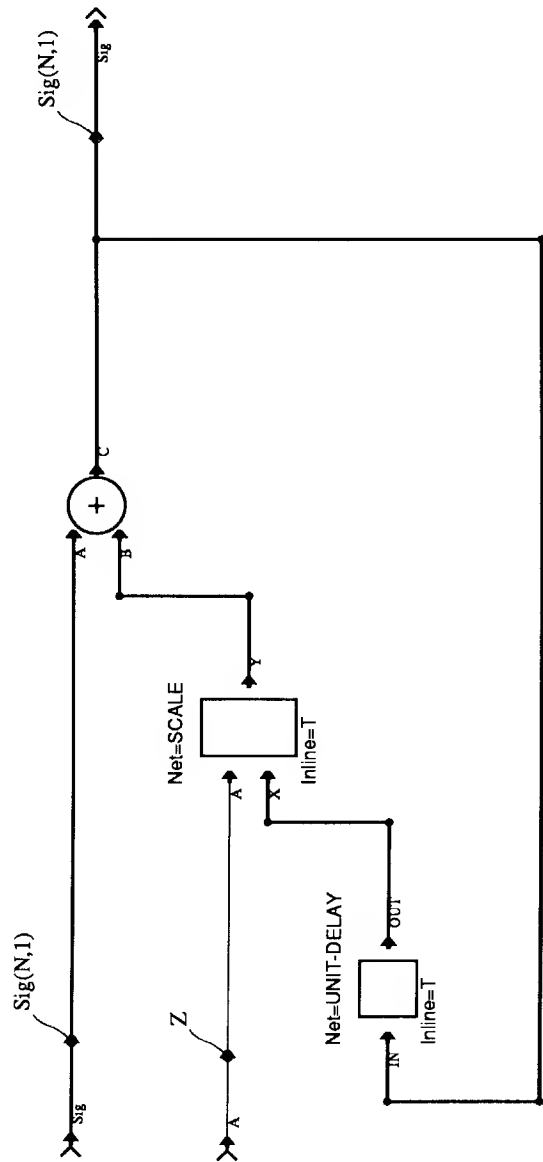


Figure A-1: A first order filter section.

FIRST-ORDER-SECTION

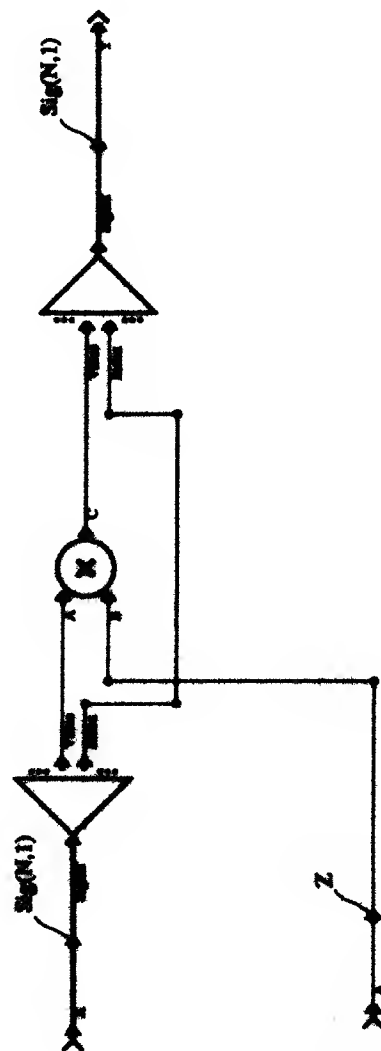
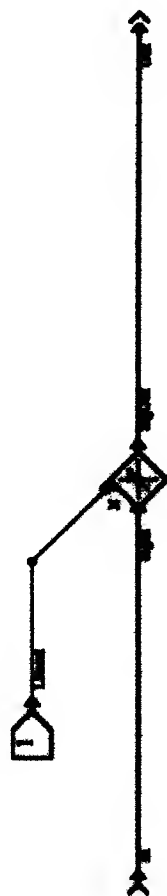


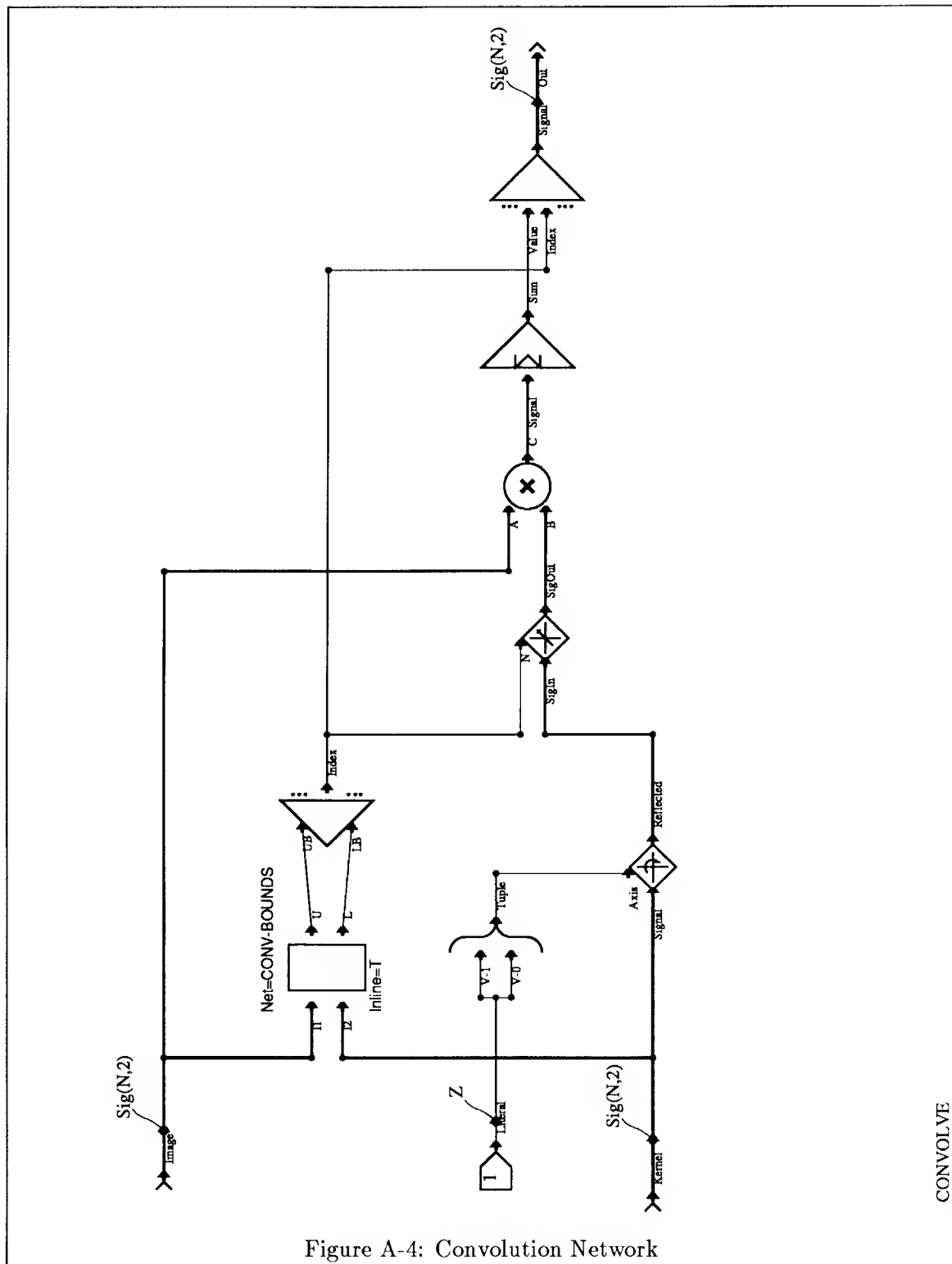
Figure A-2: Signal Scaler Network

SCALE



UNIT-DELAY

Figure A-3: Unit Delay Network

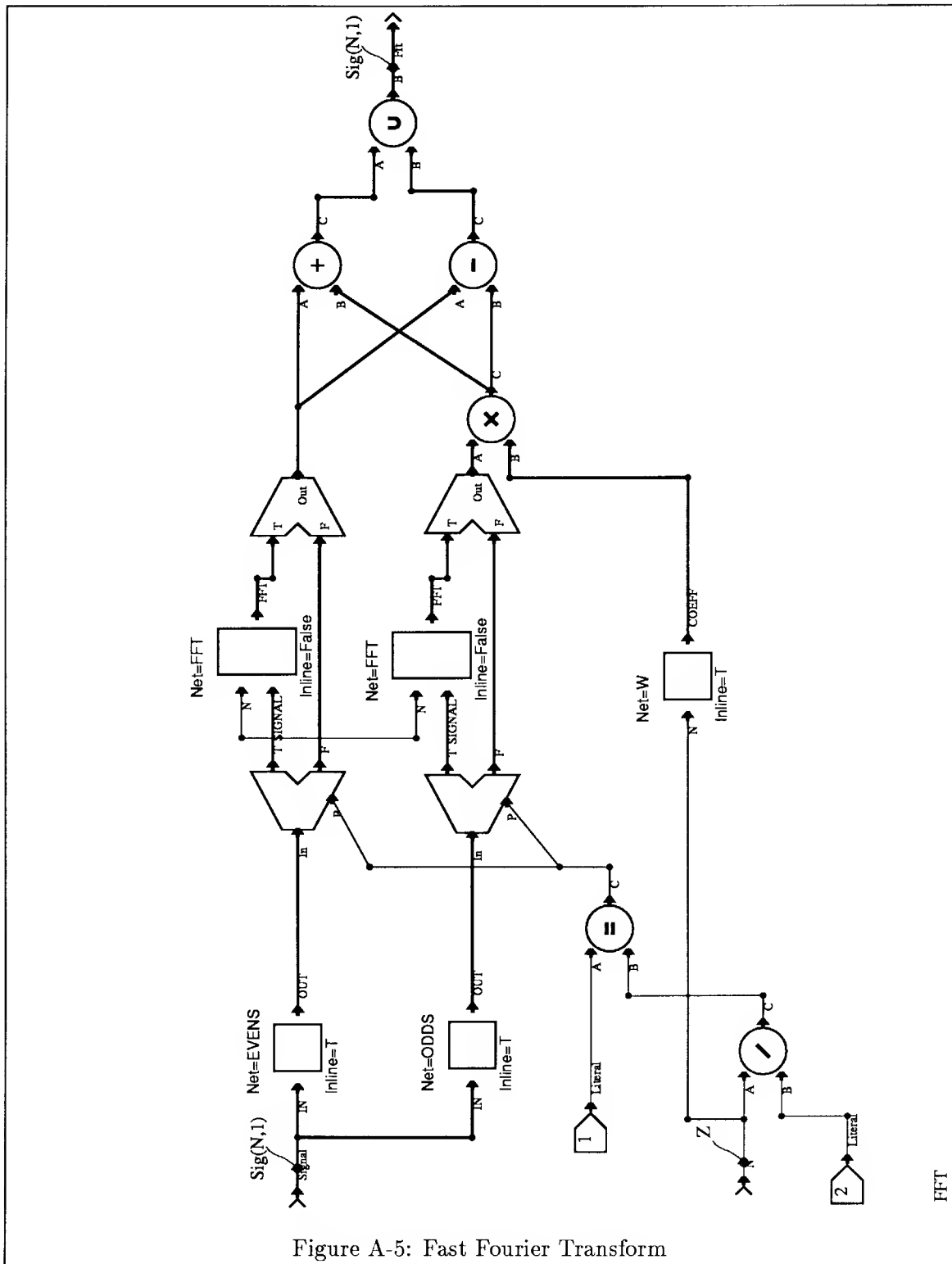


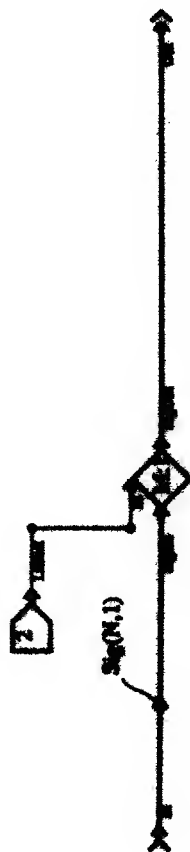
A.3 Fast Fourier Transform

Figures A-5, A-6 and A-7 show the networks for the implementation of a recursive implementation of the Fast Fourier Transform (FFT). The FFT can be reformed to be separable, and therefore expressible in an array based language, but most block-diagram languages have difficulty expressing the FFT because every element of the output depends on every element of the input signal. The FFT is not next-state-simulation realizable. Because *D-PICT* allows recursion it can express the FFT very concisely, and the “butterfly” structure of the algorithm is readily apparent in the definition.

A.4 Long Signal Filtering

Figures A-8 – A-12 contain the networks for an implementation of a filter for an input sequence with infinite explicit domain. One cannot use a direct implementation of the convolution operation with infinite sequences because it would never terminate. In some cases a FIR filter may be used, but often the filter desired may be more efficiently implemented as an FFT-convolution. This algorithm has $O(n \log n)$ behavior rather than the $O(n^2)$ growth of the straight convolution. The input signal is broken into segments, as shown in Figure A-9, and each of these segments are filtered separately. The filter, shown in Figure A-11, is implemented by applying FFT to both the segment and the system function, H , multiplying the resulting frequency signals, and performing the inverse FFT to recover the filtered segment. All of the segments are then added together to create the output signal. This procedure shows the ability of *D-PICT* to resample signals, an ability lacking in most array based and block diagram signal processing languages.





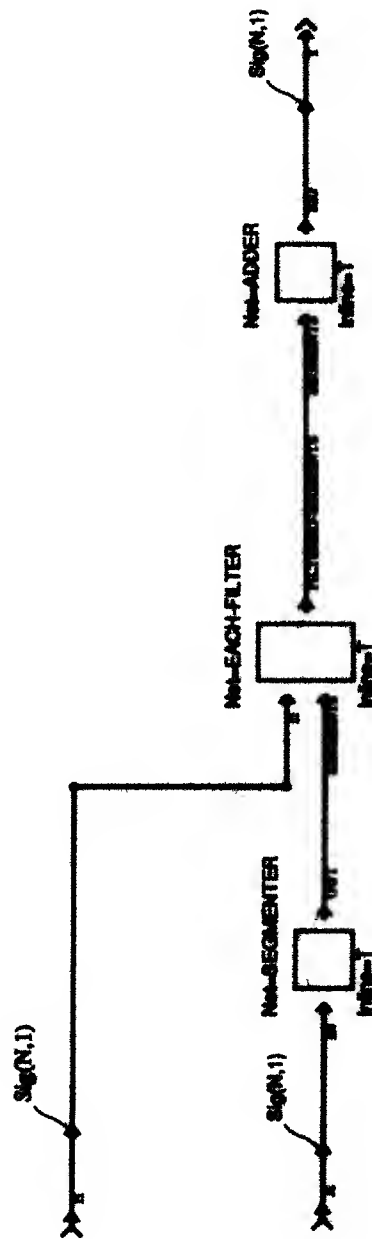
EVENS

Figure A-6: Evans Network



ODDS

Figure A-7: Odds Network



LONG-FILTER

Figure A-8: Long signal filter

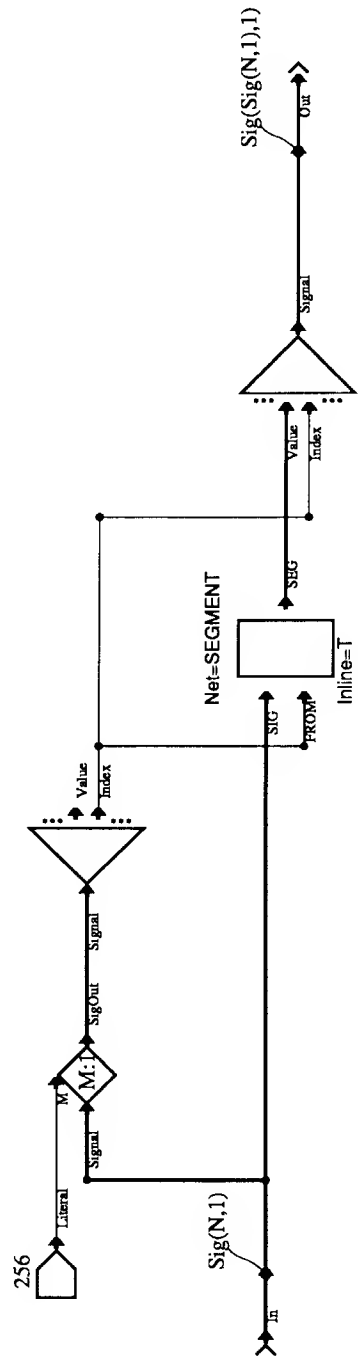


Figure A-9: Segementer Network

SEGMENTER

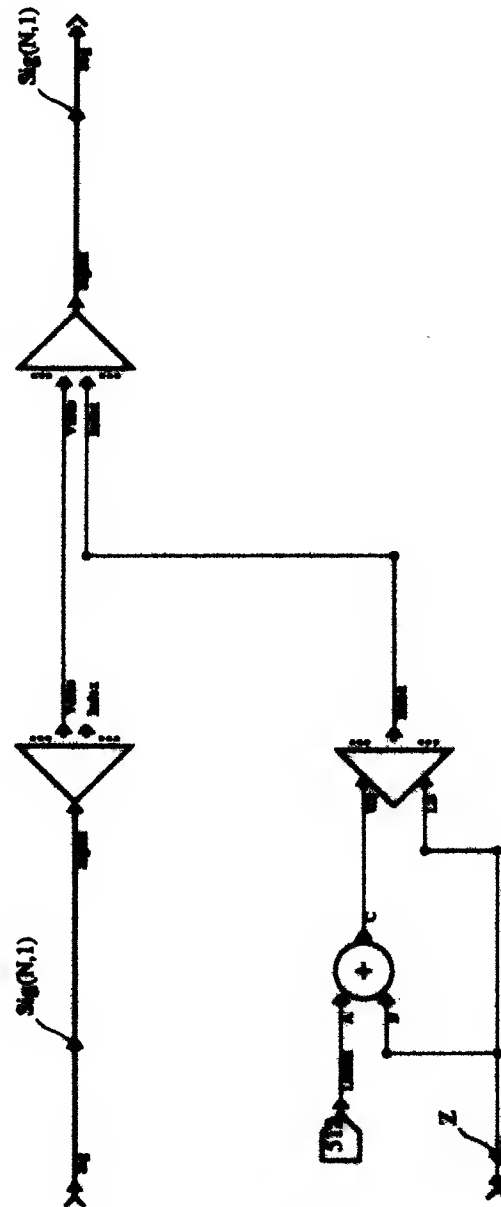
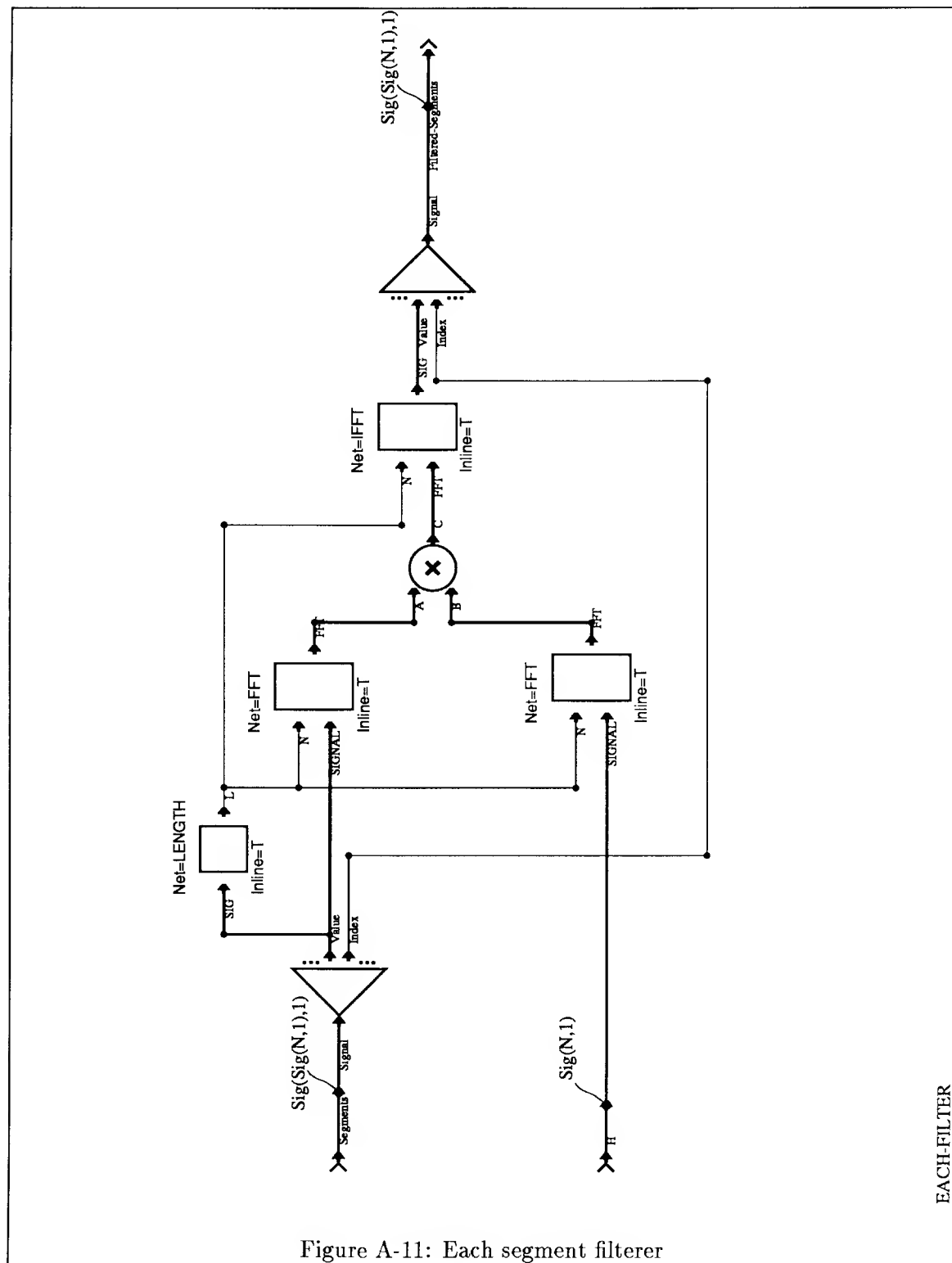


Figure A-10: Segment Network

SEGMENT



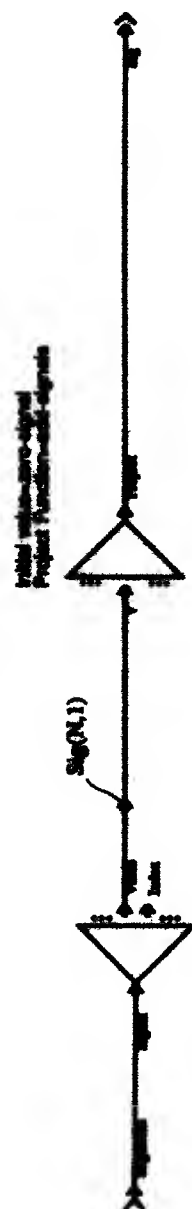


Figure A-12: Segment adder

ADDER

Bibliography

- [1] G. Abelson and J. Sussman, G. J. with Sussman. *Structure and Interpretation of Computer Programs*, page 218. The MIT Press, Cambridge, Massachusetts, 1985.
- [2] Arvind and D. E. Culler. *Dataflow Architectures*. Laboratory for Computer Science Technical Memo TM-294, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts, February 1986.
- [3] Arvind and D. E. Culler. Managing resources in a parallel machine. In *Fifth Generation Computer Architectures 1986*, pages 103–121, Elsevier Science Publishers B.V., 1986.
- [4] L. Cardelli. *Basic Polymorphic Typechecking*. Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, ca. 1984.
- [5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.
- [6] C. Clack and S. L. Peyton-Jones. Strictness analysis — a practical approach. In *Functional Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 35–49, Springer-Verlag, Berlin, September 1985.
- [7] M. Dertouzos, M. Kaliske, and K. Polzen. On-line simulation of block-diagram systems. *IEEE Transactions on Computers*, C-18, April 1969.
- [8] W. P. Dove, C. Myers, and E. E. Milios. *An Object-Oriented Signal Processing Environment: The Knowledge-Based Signal Processing Package*. Technical Report 502, MIT Research Laboratory of Electronics, Cambridge, Massachusetts, October 1984.
- [9] H. Gethöffer. SIPROL: a high-level language for digital signal processing. In *Proceedings of the 1980 IEEE Conference on Acoustics, Speech, and Signal Processing*,

1980.

- [10] D. Johnson. Signal processing software tools. In *Proceedings of the 1984 IEEE Conference on Acoustics, Speech, and Signal Processing*, pages 8.6.1 – 8.6.3, 1984.
- [11] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Association for Computing Machinery, June 1984.
- [12] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 199–203, Springer-Verlag, Berlin, September 1985.
- [13] G. E. Kopec. The integrated signal processing system ISP. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32, August 1984.
- [14] G. E. Kopec. *The Representation of Discrete-Time Signals and Systems in Programs*. PhD thesis, MIT, Cambridge, Massachusetts, 1980.
- [15] G. E. Kopec. The signal representation language SRL. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-33(4), August 1985.
- [16] G. Korn. High-speed block-diagram languages for microprocessors and minicomputers in instrumentation, control, and simulation. *Computers in Electrical Engineering*, 4, 1977.
- [17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimization. In *Conference Record of the 8th ACM Symposium on the Principles of Programming Languages*, pages 207–218, Association for Computing Machinery, January 1981.
- [18] J. I. Leivent. Dotlisp: an architecture independent high level parallel language for numeric processing. 1987. Design of a language for the MX-1 parallel processor in Group 21 at MIT Lincoln Laboratory.
- [19] P. M. D. Gray MA, PhD, MBCS. *Logic, Algebra and Databases. Ellis Horwood Series in Computers and Their Applications*, Ellis Horwood Limited, Market Cross House, Chichester, West Sussex, PO1EB, England, 1984.

- [20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
- [21] L. Morris and C. Mudge. Automatic generation of time-efficient digital signal processing software. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-26, February 1977.
- [22] R. S. Nikhil, K. Pingali, and Arvind. *Id Nouveau*. Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1986.
- [23] A. V. Oppenheim and R. W. Shafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
- [24] G. X. Ritter and P. D. Gader. Image algebra techniques for parallel image processing. *Journal for Parallel and Distributive Computing*, 1986.
- [25] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, 1982.
- [26] S. Terepin. A graphical notation for describing data flow in digital filters. In *Proceedings of the 1984 IEEE Conference on Acoustics, Speech, and Signal Processing*, pages 8.2.1 – 8.2.4, 1984.
- [27] K. R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture*. Technical Report TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1986.
- [28] K. R. Traub. *A Dataflow Compiler Substrate*. Computation Structures Group Memo 261, Laboratory for Computer Science, Massachusetts Institute of Technology, March 24 1986.
- [29] J. G. Verly, T. R. Esselman, and D. E. Dudgeon. Silhouette feature extraction in laser-radar range imagery I. Contour-based Approach. In *Iris*, November 1986. Presented at the IRIS Active Systems Specialty Group Meeting, JHU Applied Physics Laboratory.
- [30] J. G. Verly, P. L. Van Hove, R. L. Walton, and D. E. Dudgeon. Silhouette understanding system. In *Proceedings of the 1986 IEEE Conference on Acoustics, Speech, and Signal Processing*, 1986.

- [31] R. Walton. *LAMP PCP*. Informal Memo, MIT Lincoln Laboratory, September 1986.
- [32] R. Walton. *Signal Processing Primitives for LISP*. Informal Memo, MIT Lincoln Laboratory, March 1985.
- [33] R. Walton, J. Verly, and P. Van Hove. *Sketch3B*. Reference Manual, MIT Lincoln Laboratory, 1986.
- [34] B. D. Williams, J. G. Verly, T. R. Esselman, and D. E. Dudgeon. Silhouette feature extraction in laser-radar range imagery II. Intensity-cued Region-based Approach. In *Iris*, November 1986. Presented at the IRIS Active Systems Specialty Group Meeting, JHU Applied Physics Laboratory.
- [35] M. A. Zissman, G. C. O'Leary, and D. H. Johnson. A block diagram compiler for a digital signal processing mimd computer. In *Proceedings of the 1987 IEEE Conference on Acoustics, Speech, and Signal Processing*, 1987.